图灵程序设计丛书

Programming Computer Vision with Python

Python计算机视觉编程



[美] *Jan Erik Solem* 著
朱文涛 袁勇 译

O'REILLY®
人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

TURING 图灵程序设计丛书

Python计算机视觉编程

Programming Computer Vision with Python

[美] Jan Erik Solem 著
朱文涛 袁 勇 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（C I P）数据

Python计算机视觉编程 / (美) 索利姆
(Solem,J.E.) 著 ; 朱文涛, 袁勇译. -- 北京 : 人民邮
电出版社, 2014.7(2016.6重印)
(图灵程序设计丛书)
ISBN 978-7-115-35232-3

Ⅰ. ①P… Ⅱ. ①索… ②朱… ③袁… Ⅲ. ①软件工
具—程序设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2014)第067270号

内 容 提 要

本书是计算机视觉编程的权威实践指南，依赖 Python 语言讲解了基础理论与算法，并通过大量示例细致分析了对象识别、基于内容的图像搜索、光学字符识别、光流法、跟踪、三维重建、立体成像、增强现实、姿态估计、全景创建、图像分割、降噪、图像分组等技术。另外，书中附带的练习还能让读者巩固并学会应用编程知识。

本书适合的读者是：有一定编程与数学基础，想要了解计算机视觉的基本理论与算法的学生，以及计算机科学、信号处理、物理学、应用数学和统计学、神经生理学、认知科学等领域的研究人员和从业者。

◆ 著　　[美] Jan Erik Solem
译　　朱文涛　袁　勇
责任编辑　李松峰　毛倩倩
执行编辑　杨　琳
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：17.75
字数：338千字　　2014年 7 月第 1 版
印数：6 901 – 7 500册　　2016年 6 月北京第 6 次印刷

著作权合同登记号　图字：01-2014-1134号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务还是面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

目录

推荐序

计算机视觉是一门实践性很强的课程，虽然已经有了不少教科书，但大多数都只重视其中的理论和算法，很少有实践指导书。因而对于学习者而言，如果希望在实践中学习，往往需要编写大量的程序。在这方面，本书的出版可以算是一个有益的补充，相信本书将成为计算机视觉学习者的一个重要参考。

作为一本面向计算机视觉编程的书，本书涉及了这一学科中相对成熟并且被以往实践验证有效的部分典型算法，因而具有很好的实用性。例如第 2 章描述子部分选择了 Harris 角点检测器和 SIFT 描述子及其实现加以介绍；第 3 章则以全景图的创建为例，给出了 RANSAC 的实现；第 9 章图像分割中讨论了 Graph Cut 的实现等。这些方法大多数具有很好的通用性，因而为读者提供了一种实现范例。

本书的另一个特点是对介绍的单一方法，通过综合运用提升学习者灵活应用这些方法的能力。例如第 4 章给出的增强现实的例子，以及第 8 章给出的图像校正的例子。这些例子能够帮助进一步提升学习者对前述方法的感性认识。

与早期计算机视觉领域多数程序都是由 C/C++ 写就的情形不同。随着计算机硬件速度越来越快，研究者在考虑选择实现算法语言的时候会更多地考虑编写代码的效率和易用性，而不是像早年那样把算法的执行效率放在首位。这直接导致近年来越来越多的研究者选择 Python 来实现算法。与此同时，Python 的开放性使不同领域的研究者能够有机会在 Python 中加入他们需要的特性，甚至可以纳入 Python 的标准库，这也大大吸引了众多研究者对 Python 的参与。

本书的第三个特点是提供了与 OpenCV 接口的介绍。这为利用 OpenCV 中的资源提供了方便的途径。

今天在计算机视觉领域，越来越多的研究者使用 Python 开展研究。本书中文版的出版一方面能够鼓励更多的研究者采用这一语言，另一方面则为 Python 的学习者提供了一种尝试不同领域算法的机会。



陈熙霖，2014 年 4 月 21 日

前言

今天，图像和视频无处不在，在线照片分享网站和社交网络上的图像有数十亿之多。几乎对于任意可能的查询图像，搜索引擎都会给用户返回检索的图像。实际上，几乎所有手机和计算机都有内置的摄像头，所以在人们的设备中，有几 G 的图像和视频是一件很寻常的事。

计算机视觉就是用计算机编程，并设计算法来理解在这些图像中有什么。计算机视觉的有力应用有图像搜索、机器人导航、医学图像分析、照片管理等。

本书旨在为计算机视觉实战提供一个简单的切入点，让学生、研究者和爱好者充分理解其基础理论和算法。本书中的编程语言是 Python，Python 自带了很多可以免费获取的强大而便捷的图像处理、数学计算和数据挖掘模块，可以免费获取。

写作本书的时候，我遵循了以下原则。

- 鼓励探究式学习，让读者在阅读本书的时候，在计算机上跟着书中示例进行练习。
- 推广和使用免费且开源的软件，设立较低的学习门槛。显然，我们选择了 Python。
- 保持内容完整性和独立性。本书没有介绍计算机视觉的全部内容，而是完整呈现并解释所有代码。你应该能够重现这些示例，并可以直接在它们之上构建其他应用。
- 内容追求广泛而非详细，且相对于理论更注重鼓舞和激励。

总之，如果你对计算机视觉编程感兴趣，希望它能给你带来启发。

先决条件和概述

本书主要针对各种应用和问题探讨理论及算法，下面简单概括一下。

读者须知

- 基本的编程经验。你需要会使用编辑器，能够运行脚本，知道如何构建代码以及基本数据类型。熟悉 Python，或诸如 Ruby、Matlab 等其他脚本语言，这也会对你理解本书有所帮助。
- 数学基础。如果你知道矩阵、向量、矩阵乘法、标准数学函数以及导数和梯度等概念，这对于充分利用其中示例非常有益。对于一些较高级的数学例子，你可以轻松跳过。

本书内容

- 用 Python 对图像进行实战编程。
- 现实世界中各种应用背后的计算机视觉技术。
- 一些基本算法，以及如何实现并应用这些算法。

本书中的代码示例会向你展示物体识别、基于内容的图像检索、图像搜索、光学字符识别、光流、跟踪、三维重建、立体成像、增强现实、姿态估计、全景创建、图像分割、降噪、图像分组等内容。

各章概览

第 1 章“基本的图像操作和处理”介绍用来处理图像的基本工具及本书用到的核心 Python 模块，同时涵盖了很多贯穿全书的基础示例。

第 2 章“局部图像描述子”讲解检测图像兴趣点的方法，以及怎样使用它们在图像间寻找相应点和区域。

第 3 章“图像到图像的映射”描述图像间基本的变换及其计算方法。涵盖从图像扭曲到创建全景图像的示例。

第 4 章“照相机模型与增强现实”介绍如何对照相机建模、生成从三维空间到图像特征的图像投影，并估计照相机视点。

第 5 章“多视图几何”讲解如何对具有相同场景、多视图几何基本面的图像进行处理，以及怎样从图像计算三维重建。

第 6 章“图像聚类”介绍一些聚类方法，并展示如何基于相似性或内容对图像进行分组和组织。

第 7 章“图像搜索”展示如何建立有效的图像检索技术，以便能够存储图像的表示，并基于图像的视觉内容搜索图像。

第 8 章“图像内容分类”描述了图像内容分类算法，以及怎样使用它们识别图像中的物体。

第 9 章“图像分割”介绍了通过聚类、用户交互或图像模型，将图像分割成有意义区域的不同技术。

第 10 章“OpenCV”展示怎样使用常用的 OpenCV 计算机视觉库 Python 接口，以及如何处理视频及摄像头的输入。

本书结尾有参考文献。文献条目的引用用方括号表示，如 [20]。

计算机视觉简介

计算机视觉是一门对图像中信息进行自动提取的学科。信息的内容相当广泛，包括三维模型、照相机位置、目标检测与识别，以及图像内容的分组与搜索等。本书中，我们使用广义的计算机视觉概念，包括图像扭曲、降噪和增强现实等[1]。

计算机视觉有时试图模拟人类视觉，有时使用数据和统计方法，而有时几何是解决问题的关键。在本书中，我们试图对此进行全面介绍。

实用计算机视觉混合了编程、建模和数学技巧，有时很难掌握。我本着“力求简单，又不影响理解”的精神，有意用最少的理论来展示这些内容。书中对于数学知识的介绍是为了帮助读者理解算法，有些章（主要是第 4 章和第 5 章）无法避免地涉及很多数学理论。只要读者愿意，可以跳过这些数学理论，直接使用示例代码。

Python和NumPy

Python 是一门编程语言，其使用贯穿了全书的示例代码。Python 是一种简洁明了的语言，对于输入 / 输出、数字、图像及绘图都具有良好的支持。这门语言的一些特性需要我们逐渐适应，比如缩进和紧凑语法。要运行代码示例，你需要安装 Python 2.6 或之后的版本，因为只有这些版本才提供本书中用到的很多工具包。Python 3.*x* 版本与 2.*x* 版本有很多语法差异，并且不兼容 2.*x* 版本，也不兼容我们所需的工具包。

熟悉一些基本 Python 操作会更容易理解这些内容。对于 Python 初学者，我建议读一下 Mark Lutz 的书 *Learning Python*[20] 和 http://www.python.org/ 上的在线文档。

在进行计算机视觉编程的时候，我们需要在向量、矩阵的表示上进行操作，这可

注 1：这些例子生成新的图像，并且比实际地从图像中提取信息需要更多的图像处理。

以通过 Python 的 `NumPy` 模块处理；在该模块中，向量和矩阵是用 `array` 类型表示的。对于图像，我们也将采用这种类型的表示。Travis Oliphant 的免费电子书 *Guide to NumPy*[24] 是一本不错的 NumPy 参考手册；http://numpy.scipy.org/ 上的文档对于刚接触 NumPy 的读者来说是一个很好的起点。对于结果的可视化，我们会用到 `Matplotlib` 模块；而对于更高级的数学，我们会用到 `SciPy` 模块。这些就是你会用到的核心模块，详见第 1 章。

除了这些核心模块，对于某些特殊目的，比如读取 JSON 或 XML、载入并保存数据、生成图、图形编程、Web 演示、分类器等，我们还会用到很多其他免费模块。这些模块只有在特殊的应用和演示中才需要，如果你对于某种应用不感兴趣，可以跳过。

这里有必要提一下 IPython，它是一个交互式 Python 壳，使调试和实验变得更简单。对应文档及下载地址见 http://ipython.org/。

排版约定

代码如下：

```
# 一些点
x = [100,100,400,400]
y = [200,500,200,500]

# 绘制这些点
plot(x,y)
```

本书中的字体约定如下。

- 楷体
  用于定义。
- 等宽字体（`Constant width`）
  用于函数、Python 模块及代码示例，也用于控制台打印输出。

数学公式为内联式（如 $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$），或者单独居中：

$$f(\mathbf{x}) = \sum_i \mathbf{w}_i x_i + b$$

只有需要参考的时候我们才对公式进行编号。

在介绍数学知识的部分，标量使用小写字母（$s, r, \lambda, \theta\cdots$），矩阵（包括图像数组 $\boldsymbol{I}$）使用大写加粗（$\boldsymbol{A}, \boldsymbol{V}, \boldsymbol{H},\cdots$），向量则小写加粗（$\boldsymbol{t}, \boldsymbol{c},\cdots$），二维（图像）和三维中的点分别用 $\mathbf{x}=[x, y]$ 和 $\mathbf{X}[X, Y, Z]$ 表示。

使用代码示例

本书旨在帮助你完成工作。通常，你可以在程序或文档中使用本书的代码。你不必联系我们请求许可，除非你要复制本书的大量代码。例如，用本书的几段代码编写程序不需要获得许可；售卖或再分发 O'Reilly 的图书示例光盘需要获得许可；引用本书和示例代码回答问题不需要获得许可；将本书中的大量示例代码纳入产品文档中需要获得许可。

我们对你在使用时声明引用信息表示感谢，但不强制要求。引用信息通常包括标题、作者、出版商和 ISBN，例如"*Programming Computer Vision with Python* by Jan Erik Solem (O'Reilly). Copyright © 2012 Jan Erik Solem, 978-1-449-31654-9."

如果你觉得使用的代码示例超出合理引用或上述许可范围，请随时和我们联系：permissions@oreilly.com。

Safari® Books Online

Safari Books Online（www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920022923.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

致谢

感谢参与本书写作及出版的每一个人，感谢整个 O’Reilly 团队给予的帮助。特别要感谢 Andy Oram（O’Reilly）的编辑工作，以及 Paul Anagnostopoulos（Windfall Software）的高效出版工作。

很多人对于我分享在网上的书稿发表了评论。感谢 Klas Josephson 和 Håkan Ardö 就本书细致给出意见及反馈，而 Fredrik Kahl 和 Pau Gargallo 帮助进行了核实。感谢所有读者的鼓励，以及对本书文本和代码示例的改进。很多陌生人通过邮件分享了对书稿的看法，这对我而言是一个巨大的动力。

最后，感谢朋友和家人对我夜以继日写作的理解和支持。特别要感谢长久以来一直支持着我的妻子 Sara。

第 1 章

基本的图像操作和处理

本章讲解操作和处理图像的基础知识，将通过大量示例介绍处理图像所需的 Python 工具包，并介绍用于读取图像、图像转换和缩放、计算导数、画图和保存结果等的基本工具。这些工具的使用将贯穿本书的剩余章节。

1.1 PIL：Python图像处理类库

PIL（Python Imaging Library，图像处理类库）提供了通用的图像处理功能，以及大量有用的基本图像操作，比如图像缩放、裁剪、旋转、颜色转换等。PIL 是免费的，可以从 http://www.pythonware.com/products/pil/ 下载。

利用 PIL 中的函数，我们可以从大多数图像格式的文件中读取数据，然后写入最常见的图像格式文件中。PIL 中最重要的模块为 `Image`。要读取一幅图像，可以使用：

```
from PIL import Image

pil_im = Image.open('empire.jpg')
```

上述代码的返回值 `pil_im` 是一个 PIL 图像对象。

图像的颜色转换可以使用 `convert()` 方法来实现。要读取一幅图像，并将其转换成灰度图像，只需要加上 `convert('L')`，如下所示：

```
pil_im = Image.open('empire.jpg').convert('L')
```

在 PIL 文档中有一些例子，参见 http://www.pythonware.com/library/pil/handbook/index.htm。这些例子的输出结果如图 1-1 所示。



图 1-1：用 PIL 处理图像的例子

1.1.1 转换图像格式

通过 save() 方法，PIL 可以将图像保存成多种格式的文件。下面的例子从文件名列表（filelist）中读取所有的图像文件，并转换成 JPEG 格式：

```
from PIL import Image
import os

for infile in filelist:
  outfile = os.path.splitext(infile)[0] + ".jpg"
  if infile != outfile:
    try:
      Image.open(infile).save(outfile)
    except IOError:
      print "cannot convert", infile
```

PIL 的 open() 函数用于创建 PIL 图像对象，save() 方法用于保存图像到具有指定文件名的文件。除了后缀变为“.jpg”，上述代码的新文件名和原文件名相同。PIL 是个足够智能的类库，可以根据文件扩展名来判定图像的格式。PIL 函数会进行简单的检查，如果文件不是 JPEG 格式，会自动将其转换成 JPEG 格式；如果转换失败，它会在控制台输出一条报告失败的消息。

本书会处理大量图像列表。下面将创建一个包含文件夹中所有图像文件的文件名列表。首先新建一个文件，命名为 imtools.py，来存储一些经常使用的图像操作，然后将下面的函数添加进去：

```
import os
def get_imlist(path):
```

```
    """ 返回目录中所有 JPG 图像的文件名列表 """
    return [os.path.join(path,f) for f in os.listdir(path) if f.endswith('.jpg')]
```

现在，回到 PIL。

1.1.2 创建缩略图

使用 PIL 可以很方便地创建图像的缩略图。thumbnail() 方法接受一个元组参数（该参数指定生成缩略图的大小），然后将图像转换成符合元组参数指定大小的缩略图。例如，创建最长边为 128 像素的缩略图，可以使用下列命令：

```
pil_im.thumbnail((128,128))
```

1.1.3 复制和粘贴图像区域

使用 crop() 方法可以从一幅图像中裁剪指定区域：

```
box = (100,100,400,400)
region = pil_im.crop(box)
```

该区域使用四元组来指定。四元组的坐标依次是（左，上，右，下）。PIL 中指定坐标系的左上角坐标为（0，0）。我们可以旋转上面代码中获取的区域，然后使用 paste() 方法将该区域放回去，具体实现如下：

```
region = region.transpose(Image.ROTATE_180)
pil_im.paste(region,box)
```

1.1.4 调整尺寸和旋转

要调整一幅图像的尺寸，我们可以调用 resize() 方法。该方法的参数是一个元组，用来指定新图像的大小：

```
out = pil_im.resize((128,128))
```

要旋转一幅图像，可以使用逆时针方式表示旋转角度，然后调用 rotate() 方法：

```
out = pil_im.rotate(45)
```

上述例子的输出结果如图 1-1 所示。最左端是原始图像，然后是灰度图像、粘贴有旋转后裁剪图像的原始图像，最后是缩略图。

1.2 Matplotlib

我们处理数学运算、绘制图表，或者在图像上绘制点、直线和曲线时，Matplotlib 是个很好的类库，具有比 PIL 更强大的绘图功能。Matplotlib 可以绘制出高质量的图表，就像本书中的许多插图一样。Matplotlib 中的 PyLab 接口包含很多方便用户创建图像的函数。Matplotlib 是开源工具，可以从 http://matplotlib.sourceforge.net/ 免费下载。该链接中包含非常详尽的使用说明和教程。下面的例子展示了本书中需要使用的大部分函数。

1.2.1 绘制图像、点和线

尽管 Matplotlib 可以绘制出较好的条形图、饼状图、散点图等，但是对于大多数计算机视觉应用来说，仅仅需要用到几个绘图命令。最重要的是，我们想用点和线来表示一些事物，比如兴趣点、对应点以及检测出的物体。下面是用几个点和一条线绘制图像的例子：

```
from PIL import Image
from pylab import *

# 读取图像到数组中
im = array(Image.open('empire.jpg'))

# 绘制图像
imshow(im)

# 一些点
x = [100,100,400,400]
y = [200,500,200,500]

# 使用红色星状标记绘制点
plot(x,y,'r*')

# 绘制连接前两个点的线
plot(x[:2],y[:2])

# 添加标题，显示绘制的图像
title('Plotting: "empire.jpg"')
show()
```

上面的代码首先绘制出原始图像，然后在 x 和 y 列表中给定点的 x 坐标和 y 坐标上绘制出红色星状标记点，最后在两个列表表示的前两个点之间绘制一条线段（默认为蓝色）。该例子的绘制结果如图 1-2 所示。show() 命令首先打开图形用户界面（GUI），然后新建一个图像窗口。该图形用户界面会循环阻断脚本，然后暂停，直到最后一个图像窗口关闭。在每个脚本里，你只能调用一次 show() 命令，而且通常是在脚本的结尾调用。注意，在 PyLab 库中，我们约定图像的左上角为坐标原点。

图像的坐标轴是一个很有用的调试工具；但是，如果你想绘制出较美观的图像，加上下列命令可以使坐标轴不显示：

```
axis('off')
```

上面的命令将绘制出如图 1-2 右边所示的图像。

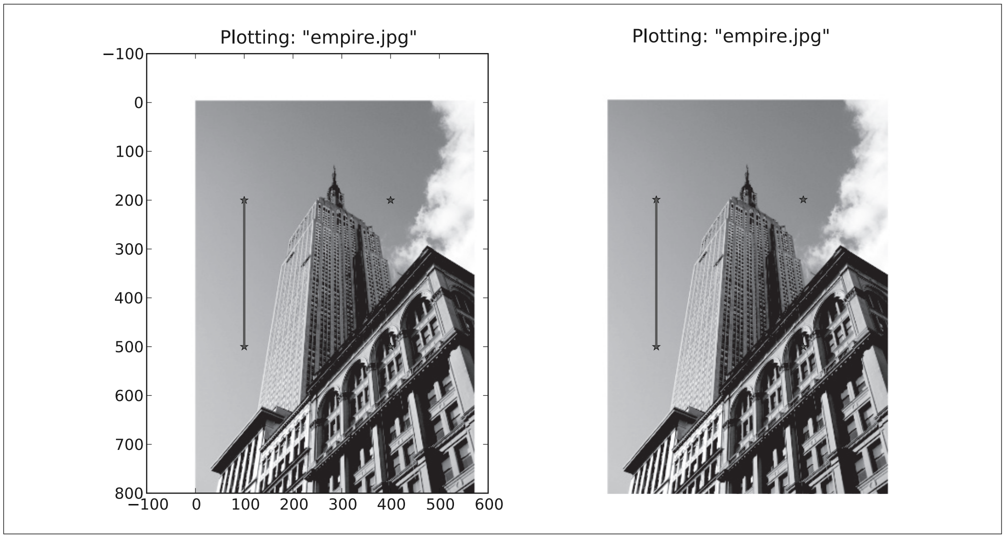


图 1-2：Matplotlib 绘图示例。带有坐标轴和不带坐标轴的包含点和一条线段的图像

在绘图时，有很多选项可以控制图像的颜色和样式。最有用的一些短命令如表 1-1、表 1-2 和表 1-3 所示。使用方法见下面的例子：

```
plot(x,y)         # 默认为蓝色实线
plot(x,y,'r*')    # 红色星状标记
plot(x,y,'go-')   # 带有圆圈标记的绿线
plot(x,y,'ks:')   # 带有正方形标记的黑色点线
```

表1-1：用PyLab库绘图的基本颜色格式命令

颜色	
'b'	蓝色
'g'	绿色
'r'	红色
'c'	青色
'm'	品红
'y'	黄色
'k'	黑色
'w'	白色

表1-2：用PyLab库绘图的基本线型格式命令

线型	
'-'	实线
'--'	虚线
':'	点线

表1-3：用PyLab库绘图的基本绘制标记格式命令

标记	
'.'	点
'o'	圆圈
's'	正方形
'*'	星形
'+'	加号
'x'	叉号

1.2.2 图像轮廓和直方图

下面来看两个特别的绘图示例：图像的轮廓和直方图。绘制图像的轮廓（或者其他二维函数的等轮廓线）在工作中非常有用。因为绘制轮廓需要对每个坐标 [*x*, *y*] 的像素值施加同一个阈值，所以首先需要将图像灰度化：

```
from PIL import Image
from pylab import *

# 读取图像到数组中
im = array(Image.open('empire.jpg').convert('L'))

# 新建一个图像
figure()
# 不使用颜色信息
gray()
# 在原点的左上角显示轮廓图像
contour(im, origin='image')
axis('equal')
axis('off')
```

像之前的例子一样，这里用 PIL 的 `convert()` 方法将图像转换成灰度图像。

图像的直方图用来表征该图像像素值的分布情况。用一定数目的小区间（bin）来指定表征像素值的范围，每个小区间会得到落入该小区间表示范围的像素数目。该（灰度）图像的直方图可以使用 `hist()` 函数绘制：

```
figure()
hist(im.flatten(),128)
show()
```

hist() 函数的第二个参数指定小区间的数目。需要注意的是，因为 hist() 只接受一维数组作为输入，所以我们在绘制图像直方图之前，必须先对图像进行压平处理。flatten() 方法将任意数组按照行优先准则转换成一维数组。图 1-3 为等轮廓线和直方图图像。

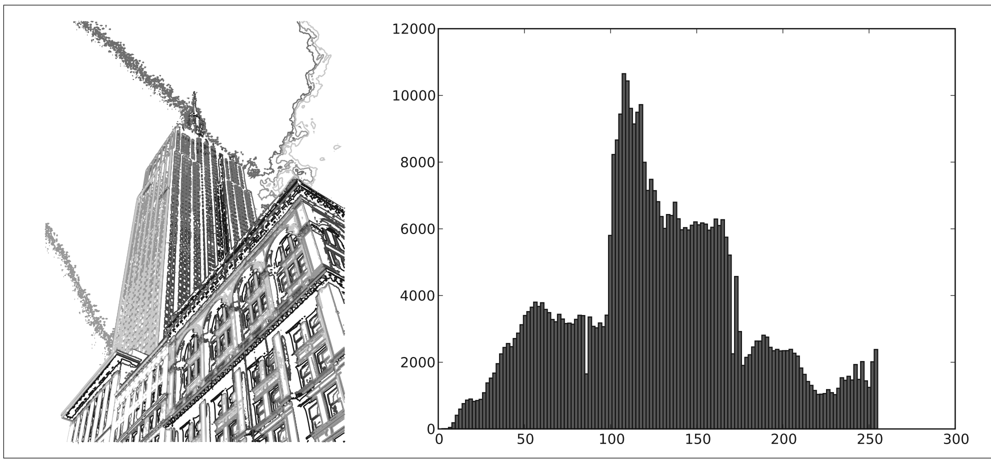


图 1-3：用 Matplotlib 绘制图像等轮廓线和直方图

1.2.3　交互式标注

有时用户需要和某些应用交互，例如在一幅图像中标记一些点，或者标注一些训练数据。PyLab 库中的 ginput() 函数就可以实现交互式标注。下面是一个简短的例子：

```
from PIL import Image
from pylab import *

im = array(Image.open('empire.jpg'))
imshow(im)
print 'Please click 3 points'
x = ginput(3)
print 'you clicked:',x
show()
```

上面的脚本首先绘制一幅图像，然后等待用户在绘图窗口的图像区域点击三次。程序将这些点击的坐标 $[x, y]$ 自动保存在 x 列表里。

1.3 NumPy

NumPy（http://www.scipy.org/NumPy/）是非常有名的 Python 科学计算工具包，其中包含了大量有用的思想，比如数组对象（用来表示向量、矩阵、图像等）以及线性代数函数。NumPy 中的数组对象几乎贯穿用于本书的所有例子中 [1] 数组对象可以帮助你实现数组中重要的操作，比如矩阵乘积、转置、解方程系统、向量乘积和归一化，这为图像变形、对变化进行建模、图像分类、图像聚类等提供了基础。

NumPy 可以从 http://www.scipy.org/Download 免费下载，在线说明文档（http://docs.scipy.org/doc/numpy/）包含了你可能遇到的大多数问题的答案。关于 NumPy 的更多内容，请参考开源书籍 [24]。

1.3.1 图像数组表示

在先前的例子中，当载入图像时，我们通过调用 array() 方法将图像转换成 NumPy 的数组对象，但当时并没有进行详细介绍。NumPy 中的数组对象是多维的，可以用来表示向量、矩阵和图像。一个数组对象很像一个列表（或者是列表的列表），但是数组中所有的元素必须具有相同的数据类型。除非创建数组对象时指定数据类型，否则数据类型会按照数据的类型自动确定。

对于图像数据，下面的例子阐述了这一点：

```
im = array(Image.open('empire.jpg'))
print im.shape, im.dtype

im = array(Image.open('empire.jpg').convert('L'),'f')
print im.shape, im.dtype
```

控制台输出结果如下所示：

```
(800, 569, 3) uint8
(800, 569) float32
```

每行的第一个元组表示图像数组的大小（行、列、颜色通道），紧接着的字符串表示数组元素的数据类型。因为图像通常被编码成无符号八位整数（uint8），所以在第一种情况下，载入图像并将其转换到数组中，数组的数据类型为“uint8”。在第二种情况下，对图像进行灰度化处理，并且在创建数组时使用额外的参数“f”；该参数将数据类型转换为浮点型。关于更多数据类型选项，可以参考图书 [24]。注意，由于灰度图像没有颜色信息，所以在形状元组中，它只有两个数值。

注 1：PyLab 实际上包含 NumPy 的一些内容，如数组类型。这也是我们能够在 1.2 节使用数组类型的原因。

数组中的元素可以使用下标访问。位于坐标 i、j，以及颜色通道 k 的像素值可以像下面这样访问：

```
value = im[i,j,k]
```

多个数组元素可以使用数组切片方式访问。*切片方式*返回的是以指定间隔下标访问该数组的元素值。下面是有关灰度图像的一些例子：

```
im[i,:] = im[j,:]        # 将第 j 行的数值赋值给第 i 行
im[:,i] = 100            # 将第 i 列的所有数值设为 100
im[:100,:50].sum()       # 计算前 100 行、前 50 列所有数值的和
im[50:100,50:100]        # 50~100 行，50~100 列（不包括第 100 行和第 100 列）
im[i].mean()             # 第 i 行所有数值的平均值
im[:,-1]                 # 最后一列
im[-2,:] (or im[-2])     # 倒数第二行
```

注意，示例仅仅使用一个下标访问数组。如果仅使用一个下标，则该下标为行下标。注意，在最后几个例子中，负数切片表示从最后一个元素逆向计数。我们将会频繁地使用切片技术访问像素值，这也是一个很重要的思想。

我们有很多操作和方法来处理数组对象。本书将在使用到的地方逐一介绍。你可以查阅在线文档或者开源图书 [24] 获取更多信息。

1.3.2 灰度变换

将图像读入 NumPy 数组对象后，我们可以对它们执行任意数学操作。一个简单的例子就是图像的灰度变换。考虑任意函数 f，它将 0...255 区间（或者 0...1 区间）映射到自身（意思是说，输出区间的范围和输入区间的范围相同）。下面是关于灰度变换的一些例子：

```
from PIL import Image
from numpy import *

im = array(Image.open('empire.jpg').convert('L'))

im2 = 255 - im # 对图像进行反相处理

im3 = (100.0/255) * im + 100 # 将图像像素值变换到 100...200 区间

im4 = 255.0 * (im/255.0)**2 # 对图像像素值求平方后得到的图像
```

第一个例子将灰度图像进行反相处理；第二个例子将图像的像素值变换到 100...200 区间；第三个例子对图像使用二次函数变换，使较暗的像素值变得更小。图 1-4 为所使用的变换函数图像。图 1-5 是输出的图像结果。你可以使用下面的命令查看图

像中的最小和最大像素值：

```
print int(im.min()), int(im.max())
```

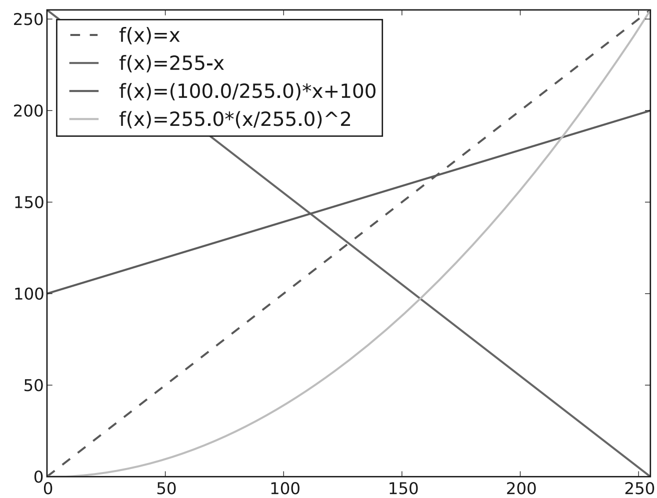


图 1-4：灰度变换示例。三个例子中所使用函数的图像，其中虚线表示恒等变换



图 1-5：灰度变换。对图像应用图 1-4 中的函数：$f(x)=255-x$ 对图像进行反相处理（左）；$f(x)=(100/255)x+100$ 对图像进行变换（中）；$f(x)=255(x/255)^2$ 对图像做二次变换（右）

如果试着对上面例子查看最小值和最大值，可以得到下面的输出结果：

```
2 255
0 253
100 200
0 255
```

`array()` 变换的相反操作可以使用 PIL 的 `fromarray()` 函数完成：

```
pil_im = Image.fromarray(im)
```

如果你通过一些操作将“uint8”数据类型转换为其他数据类型，比如之前例子中的 im3 或者 im4，那么在创建 PIL 图像之前，需要将数据类型转换回来：

```
pil_im = Image.fromarray(uint8(im))
```

如果你并不十分确定输入数据的类型，安全起见，应该先转换回来。注意，NumPy 总是将数组数据类型转换成能够表示数据的“最低”数据类型。对浮点数做乘积或除法操作会使整数类型的数组变成浮点类型。

1.3.3 图像缩放

NumPy 的数组对象是我们处理图像和数据的主要工具。想要对图像进行缩放处理没有现成简单的方法。我们可以使用之前 PIL 对图像对象转换的操作，写一个简单的用于图像缩放的函数。把下面的函数添加到 imtool.py 文件里：

```
def imresize(im,sz):
  """ 使用 PIL 对象重新定义图像数组的大小 """
  pil_im = Image.fromarray(uint8(im))

  return array(pil_im.resize(sz))
```

我们将会在接下来的内容中使用这个函数。

1.3.4 直方图均衡化

图像灰度变换中一个非常有用的例子就是*直方图均衡化*。直方图均衡化是指将一幅图像的灰度直方图变平，使变换后的图像中每个灰度值的分布概率都相同。在对图像做进一步处理之前，直方图均衡化通常是对图像灰度值进行归一化的一个非常好的方法，并且可以增强图像的对比度。

在这种情况下，直方图均衡化的变换函数是图像中像素值的*累积分布函数*（cumulative distribution function，简写为 cdf，将像素值的范围映射到目标范围的归一化操作）。

下面的函数是直方图均衡化的具体实现。将这个函数添加到 imtool.py 里：

```
def histeq(im,nbr_bins=256):
  """ 对一幅灰度图像进行直方图均衡化 """

  # 计算图像的直方图
  imhist,bins = histogram(im.flatten(),nbr_bins,normed=True)
  cdf = imhist.cumsum() # 累积分布函数
  cdf = 255 * cdf / cdf[-1] # 归一化
```

```
    # 使用累积分布函数的线性插值，计算新的像素值
    im2 = interp(im.flatten(),bins[:-1],cdf)

    return im2.reshape(im.shape), cdf
```

该函数有两个输入参数，一个是灰度图像，一个是直方图中使用小区间的数目。函数返回直方图均衡化后的图像，以及用来做像素值映射的累积分布函数。注意，函数中使用到累积分布函数的最后一个元素（下标为 –1），目的是将其归一化到 0...1 范围。你可以像下面这样使用该函数：

```
from PIL import Image
from numpy import *

im = array(Image.open('AquaTermi_lowcontrast.jpg').convert('L'))
im2,cdf = imtools.histeq(im)
```

图 1-6 和图 1-7 为上面直方图均衡化例子的结果。上面一行显示的分别是直方图均衡化之前和之后的灰度直方图，以及累积概率分布函数映射图像。可以看到，直方图均衡化后图像的对比度增强了，原先图像灰色区域的细节变得清晰。

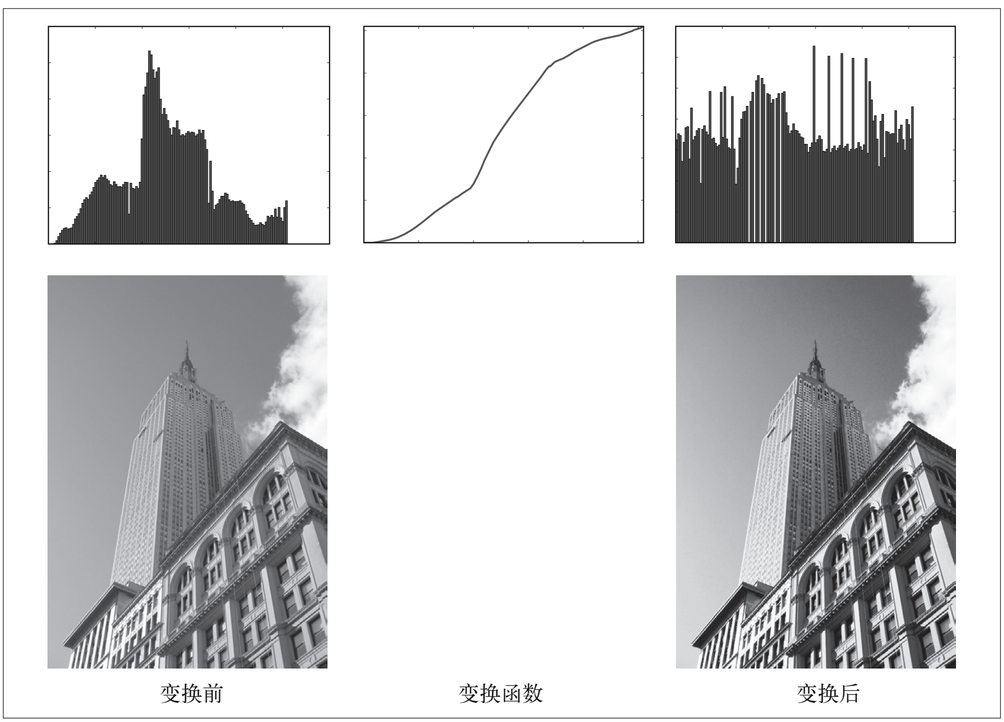


图 1-6：直方图均衡化示例。左侧为原始图像和直方图，中间图为灰度变换函数，右侧为直方图均衡化后的图像和相应直方图

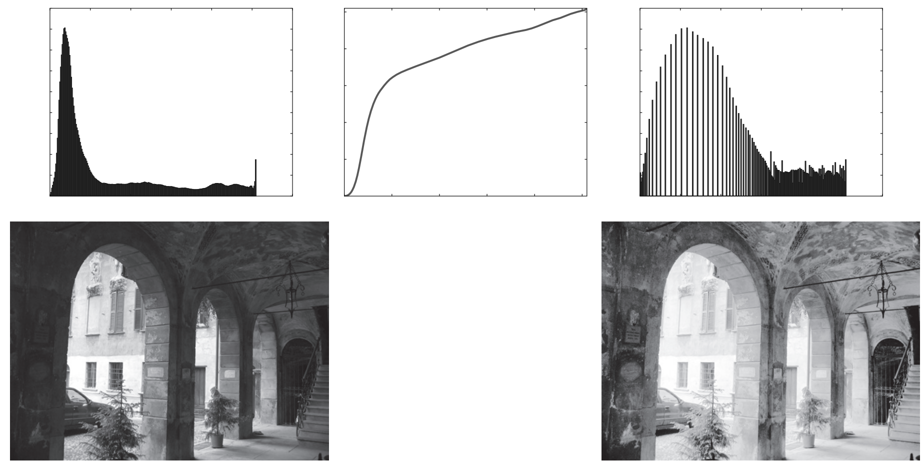


图 1-7：直方图均衡化示例。左侧为原始图像和直方图，中间图为灰度变换函数，右侧为直方图均衡化后的图像和相应直方图

1.3.5　图像平均

图像平均操作是减少图像噪声的一种简单方式，通常用于艺术特效。我们可以简单地从图像列表中计算出一幅平均图像。假设所有的图像具有相同的大小，我们可以将这些图像简单地相加，然后除以图像的数目，来计算平均图像。下面的函数可以用于计算平均图像，将其添加到 imtool.py 文件里：

```
def compute_average(imlist):
  """ 计算图像列表的平均图像 """

  # 打开第一幅图像，将其存储在浮点型数组中
  averageim = array(Image.open(imlist[0]), 'f')

  for imname in imlist[1:]:
    try:
      averageim += array(Image.open(imname))
    except:
      print imname + '...skipped'
  averageim /= len(imlist)

  # 返回 uint8 类型的平均图像
  return array(averageim, 'uint8')
```

该函数包括一些基本的异常处理技巧，可以自动跳过不能打开的图像。我们还可以使用 mean() 函数计算平均图像。mean() 函数需要将所有的图像堆积到一个数组中；也就是说，如果有很多图像，该处理方式需要占用很多内存。我们将会在下一节中使用该函数。

1.3.6 图像的主成分分析（PCA）

PCA（Principal Component Analysis，主成分分析）是一个非常有用的降维技巧。它可以在使用尽可能少维数的前提下，尽量多地保持训练数据的信息，在此意义上是一个最佳技巧。即使是一幅 100×100 像素的小灰度图像，也有 10 000 维，可以看成 10 000 维空间中的一个点。一兆像素的图像具有百万维。由于图像具有很高的维数，在许多计算机视觉应用中，我们经常使用降维操作。PCA 产生的投影矩阵可以被视为将原始坐标变换到现有的坐标系，坐标系中的各个坐标按照重要性递减排列。

为了对图像数据进行 PCA 变换，图像需要转换成一维向量表示。我们可以使用 NumPy 类库中的 flatten() 方法进行变换。

将变平的图像堆积起来，我们可以得到一个矩阵，矩阵的一行表示一幅图像。在计算主方向之前，所有的行图像按照平均图像进行了中心化。我们通常使用 SVD（Singular Value Decomposition，奇异值分解）方法来计算主成分；但当矩阵的维数很大时，SVD 的计算非常慢，所以此时通常不使用 SVD 分解。下面就是 PCA 操作的代码：

```
from PIL import Image
from numpy import *

def pca(X):
  """ 主成分分析：
    输入：矩阵 X，其中该矩阵中存储训练数据，每一行为一条训练数据
    返回：投影矩阵（按照维度的重要性排序）、方差和均值 """

  # 获取维数
  num_data,dim = X.shape

  # 数据中心化
  mean_X = X.mean(axis=0)
  X = X - mean_X

if dim>num_data:
  # PCA- 使用紧致技巧
  M = dot(X,X.T) # 协方差矩阵
  e,EV = linalg.eigh(M) # 特征值和特征向量
  tmp = dot(X.T,EV).T # 这就是紧致技巧
  V = tmp[::-1] # 由于最后的特征向量是我们所需要的，所以需要将其逆转
  S = sqrt(e)[::-1] # 由于特征值是按照递增顺序排列的，所以需要将其逆转
  for i in range(V.shape[1]):
    V[:,i] /= S
else:
  # PCA- 使用 SVD 方法
  U,S,V = linalg.svd(X)
  V = V[:num_data] # 仅仅返回前 nun_data 维的数据才合理

# 返回投影矩阵、方差和均值
return V,S,mean_X
```

该函数首先通过减去每一维的均值将数据中心化，然后计算协方差矩阵对应最大特征值的特征向量，此时可以使用简明的技巧或者 SVD 分解。这里我们使用了 range() 函数，该函数的输入参数为一个整数 n，函数返回整数 $0...(n-1)$ 的一个列表。你也可以使用 arange() 函数来返回一个数组，或者使用 xrange() 函数返回一个产生器（可能会提升速度）。我们在本书中贯穿使用 range() 函数。

如果数据个数小于向量的维数，我们不用 SVD 分解，而是计算维数更小的协方差矩阵 XX^T 的特征向量。通过仅计算对应前 k（k 是降维后的维数）最大特征值的特征向量，可以使上面的 PCA 操作更快。由于篇幅所限，有兴趣的读者可以自行探索。矩阵 $\boldsymbol{V}$ 的每行向量都是正交的，并且包含了训练数据方差依次减少的坐标方向。

我们接下来对字体图像进行 PCA 变换。fontimages.zip 文件包含采用不同字体的字符 a 的缩略图。所有的 2359 种字体可以免费下载 [1]。假定这些图像的名称保存在列表 imlist 中，跟之前的代码一起保存传在 pca.py 文件中，我们可以使用下面的脚本计算图像的主成分：

```
from PIL import Image
from numpy import *
from pylab import *
import pca

im = array(Image.open(imlist[0])) # 打开一幅图像，获取其大小
m,n = im.shape[0:2] # 获取图像的大小
imnbr = len(imlist) # 获取图像的数目

# 创建矩阵，保存所有压平后的图像数据
immatrix = array([array(Image.open(im)).flatten()
              for im in imlist],'f')

# 执行 PCA 操作
V,S,immean = pca.pca(immatrix)

# 显示一些图像（均值图像和前 7 个模式）
figure()
gray()
subplot(2,4,1)
imshow(immean.reshape(m,n))
for i in range(7):
  subplot(2,4,i+2)
  imshow(V[i].reshape(m,n))

show()
```

注意，图像需要从一维表示重新转换成二维图像；可以使用 reshape() 函数。如图 1-8 所示，运行该例子会在一个绘图窗口中显示 8 个图像。这里我们使用了 PyLab 库的 subplot() 函数在一个窗口中放置多个图像。

注 1：免费字体图像库由 Martin Solli 收集并上传（http://webstaff.itn.liu.se/~marso/）。

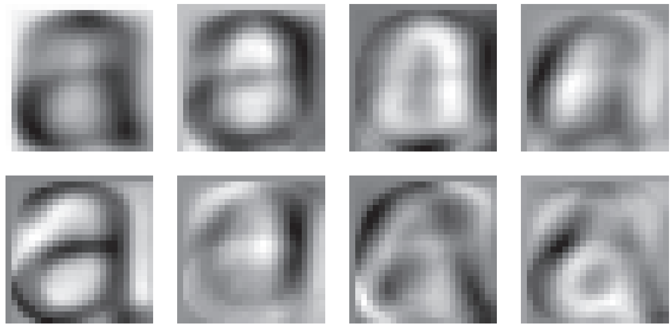


图 1-8：平均图像（左上）和前 7 个模式（具有最大方差的方向模式）

1.3.7 使用pickle模块

如果想要保存一些结果或者数据以方便后续使用，Python 中的 pickle 模块非常有用。pickle 模块可以接受几乎所有的 Python 对象，并且将其转换成字符串表示，该过程叫做封装（pickling）。从字符串表示中重构该对象，称为拆封（unpickling）。这些字符串表示可以方便地存储和传输。

我们来看一个例子。假设想要保存上一节字体图像的平均图像和主成分，可以这样来完成：

```
# 保存均值和主成分数据
f = open('font_pca_modes.pkl', 'wb')
pickle.dump(immean,f)
pickle.dump(V,f)
f.close()
```

在上述例子中，许多对象可以保存到同一个文件中。pickle 模块中有很多不同的协议可以生成 .pkl 文件；如果不确定的话，最好以二进制文件的形式读取和写入。在其他 Python 会话中载入数据，只需要如下使用 load() 方法：

```
# 载入均值和主成分数据
f = open('font_pca_modes.pkl', 'rb')
immean = pickle.load(f)
V = pickle.load(f)
f.close()
```

注意，载入对象的顺序必须和先前保存的一样。Python 中有个用 C 语言写的优化版本，叫做 cpickle 模块，该模块和标准 pickle 模块完全兼容。关于 pickle 模块的更多内容，参见 pickle 模块文档页 http://docs.python.org/library/pickle.html。

在本书接下来的章节中，我们将使用 with 语句处理文件的读写操作。这是 Python 2.5 引入的思想，可以自动打开和关闭文件（即使在文件打开时发生错误）。下面的例子使用 with() 来实现保存和载入操作：

```
# 打开文件并保存
with open('font_pca_modes.pkl', 'wb') as f:
  pickle.dump(immean,f)
  pickle.dump(V,f)
```

和

```
# 打开文件并载入
with open('font_pca_modes.pkl', 'rb') as f:
  immean = pickle.load(f)
  V = pickle.load(f)
```

上面的例子乍看起来可能很奇怪，但 with() 确实是个很有用的思想。如果你不喜欢它，可以使用之前的 open 和 close 函数。

作为 pickle 的一种替代方式，NumPy 具有读写文本文件的简单函数。如果数据中不包含复杂的数据结构，比如在一幅图像上点击的点列表，NumPy 的读写函数会很有用。保存一个数组 x 到文件中，可以使用：

```
savetxt('test.txt',x,'%i')
```

最后一个参数表示应该使用整数格式。类似地，读取可以使用：

```
x = loadtxt('test.txt')
```

你可以从在线文档 http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html 了解更多内容。

最后，NumPy 有专门用于保存和载入数组的函数。你可以在上面的在线文档里查看关于 save() 和 load() 的更多内容。

1.4 SciPy

SciPy（http://scipy.org/）是建立在 NumPy 基础上，用于数值运算的开源工具包。SciPy 提供很多高效的操作，可以实现数值积分、优化、统计、信号处理，以及对我们来说最重要的图像处理功能。接下来，本节会介绍 SciPy 中大量有用的模块。SciPy 是个开源工具包，可以从 http://scipy.org/Download 下载。

1.4.1　图像模糊

图像的高斯模糊是非常经典的图像卷积例子。本质上，图像模糊就是将（灰度）图像 $\boldsymbol{I}$ 和一个高斯核进行卷积操作：

$$\boldsymbol{I}_\sigma = \boldsymbol{I} * G_\sigma$$

其中 * 表示卷积操作；G_σ 是标准差为 σ 的二维高斯核，定义为 :

$$G_\sigma = \frac{1}{2\pi\sigma^2} \mathrm{e}^{-(x^2+y^2)/2\sigma^2}$$

高斯模糊通常是其他图像处理操作的一部分，比如图像插值操作、兴趣点计算以及很多其他应用。

SciPy 有用来做滤波操作的 scipy.ndimage.filters 模块。该模块使用快速一维分离的方式来计算卷积。你可以像下面这样来使用它：

```
from PIL import Image
from numpy import *
from scipy.ndimage import filters

im = array(Image.open('empire.jpg').convert('L'))
im2 = filters.gaussian_filter(im,5)
```

上面 guassian_filter() 函数的最后一个参数表示标准差。

图 1-9 显示了随着 σ 的增加，一幅图像被模糊的程度。σ 越大，处理后的图像细节丢失越多。如果打算模糊一幅彩色图像，只需简单地对每一个颜色通道进行高斯模糊：

```
im = array(Image.open('empire.jpg'))
im2 = zeros(im.shape)
for i in range(3):
  im2[:,:,i] = filters.gaussian_filter(im[:,:,i],5)
im2 = uint8(im2)
```

在上面的脚本中，最后并不总是需要将图像转换成 uint8 格式，这里只是将像素值用八位来表示。我们也可以使用：

```
im2 = array(im2,'uint8')
```

来完成转换。

关于该模块更多的内容以及不同参数的选择，请查看 http://docs.scipy.org/doc /scipy/reference/ndimage.html 上 SciPy 文档中的 scipy.ndimage 部分。

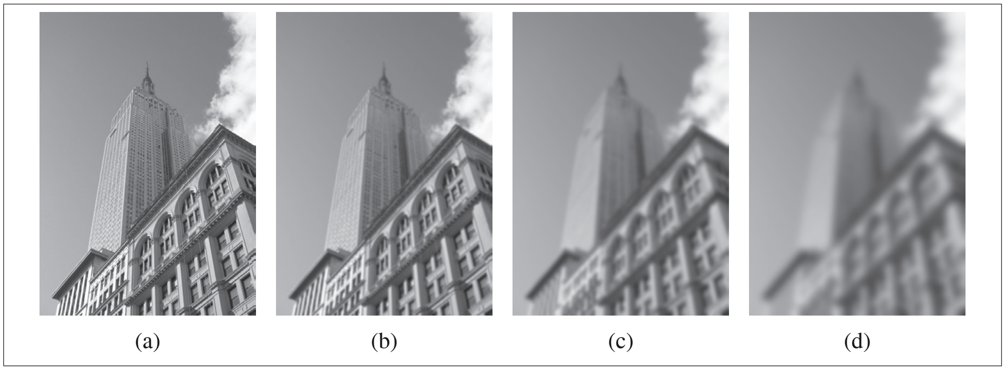


图 1-9：使用 scipy.ndimage.filters 模块进行高斯模糊：（a）原始灰度图像；（b）使用 σ=2 的高斯滤波器；（c）使用 σ=5 的高斯滤波器；（d）使用 σ=10 的高斯滤波器

1.4.2 图像导数

整本书中可以看到，在很多应用中图像强度的变化情况是非常重要的信息。强度的变化可以用灰度图像 $\boldsymbol{I}$（对于彩色图像，通常对每个颜色通道分别计算导数）的 x 和 y 方向导数 $\boldsymbol{I}_x$ 和 $\boldsymbol{I}_y$ 进行描述。

图像的梯度向量为 $\nabla \boldsymbol{I} = [\boldsymbol{I}_x, \boldsymbol{I}_y]^T$。梯度有两个重要的属性，一是梯度的大小：

$$|\nabla \boldsymbol{I}| = \sqrt{\boldsymbol{I}_x^2 + \boldsymbol{I}_y^2}$$

它描述了图像强度变化的强弱，一是梯度的角度：

$$\alpha = \text{arctan2}(\boldsymbol{I}_y, \boldsymbol{I}_x)$$

描述了图像中在每个点（像素）上强度变化最大的方向。NumPy 中的 arctan2() 函数返回弧度表示的有符号角度，角度的变化区间为 $-\pi...\pi$。

我们可以用离散近似的方式来计算图像的导数。图像导数大多数可以通过卷积简单地实现：

$$\boldsymbol{I}_x = \boldsymbol{I} * D_x \text{ 和 } \boldsymbol{I}_y = \boldsymbol{I} * D_y$$

对于 D_x 和 D_y，通常选择 Prewitt 滤波器：

$$D_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{和} D_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

或者 Sobel 滤波器：

$$D_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{和}\ D_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

这些导数滤波器可以使用 `scipy.ndimage.filters` 模块的标准卷积操作来简单地实现，例如：

```
from PIL import Image
from numpy import *
from scipy.ndimage import filters

im = array(Image.open('empire.jpg').convert('L'))

# Sobel 导数滤波器
imx = zeros(im.shape)
filters.sobel(im,1,imx)

imy = zeros(im.shape)
filters.sobel(im,0,imy)

magnitude = sqrt(imx**2+imy**2)
```

上面的脚本使用 Sobel 滤波器来计算 x 和 y 的方向导数，以及梯度大小。`sobel()` 函数的第二个参数表示选择 x 或者 y 方向导数，第三个参数保存输出的变量。图 1-10 显示了用 Sobel 滤波器计算出的导数图像。在两个导数图像中，正导数显示为亮的像素，负导数显示为暗的像素。灰色区域表示导数的值接近于零。

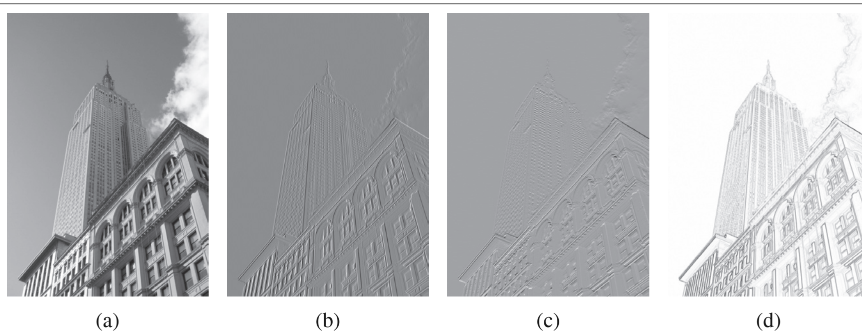


图 1-10：使用 Sobel 导数滤波器计算导数图像：（a）原始灰度图像；（b）x 导数图像；（c）y 导数图像；（d）梯度大小图像

上述计算图像导数的方法有一些缺陷：在该方法中，滤波器的尺度需要随着图像分辨率的变化而变化。为了在图像噪声方面更稳健，以及在任意尺度上计算导数，我们可以使用高斯导数滤波器：

$$\boldsymbol{I}_x=\boldsymbol{I}*G_{\sigma x} \text{ 和 } \boldsymbol{I}_y=\boldsymbol{I}*G_{\sigma y}$$

其中，$G_{\sigma x}$ 和 $G_{\sigma y}$ 表示 G_σ 在 x 和 y 方向上的导数，G_σ 为标准差为 σ 的高斯函数。

我们之前用于模糊的 `filters.gaussian_filter()` 函数可以接受额外的参数，用来计算高斯导数。可以简单地按照下面的方式来处理：

```
sigma = 5 # 标准差

imx = zeros(im.shape)
filters.gaussian_filter(im, (sigma,sigma), (0,1), imx)

imy = zeros(im.shape)
filters.gaussian_filter(im, (sigma,sigma), (1,0), imy)
```

该函数的第三个参数指定对每个方向计算哪种类型的导数，第二个参数为使用的标准差。你可以查看相应文档了解详情。图 1-11 显示了不同尺度下的导数图像和梯度大小。你可以和图 1-9 中做相同尺度模糊的图像做比较。

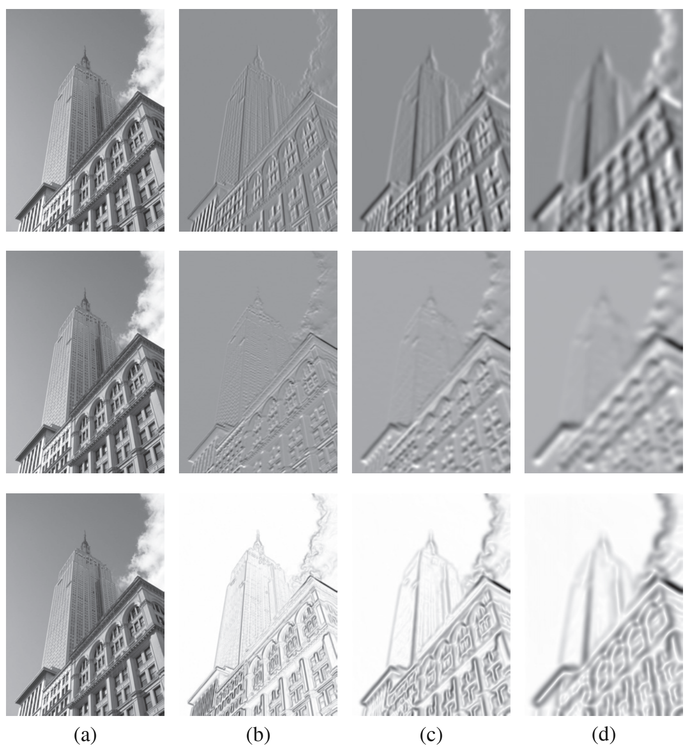


图 1-11：使用高斯导数计算图像导数：x 导数图像（上），y 导数图像（中），以及梯度大小图像（下）；（a）为原始灰度图像，（b）为使用 σ=2 的高斯导数滤波器处理后的图像，（c）为使用 σ=5 的高斯导数滤波器处理后的图像，（d）为使用 σ=10 的高斯导数滤波器处理后的图像

1.4.3 形态学：对象计数

形态学（或数学形态学）是度量和分析基本形状的图像处理方法的基本框架与集合。形态学通常用于处理二值图像，但是也能够用于灰度图像。*二值图像*是指图像的每个像素只能取两个值，通常是 0 和 1。二值图像通常是，在计算物体的数目，或者度量其大小时，对一幅图像进行阈值化后的结果。你可以从 http://en.wikipedia.org/wiki/Mathematical_morphology 大体了解形态学及其处理图像的方式。

scipy.ndimage 中的 morphology 模块可以实现形态学操作。你可以使用 scipy.ndimage 中的 measurements 模块来实现二值图像的计数和度量功能。下面通过一个简单的例子介绍如何使用它们。

考虑在图 1-12a[1] 里的二值图像，计算该图像中的对象个数可以通过下面的脚本实现：

```
from scipy.ndimage import measurements,morphology

# 载入图像，然后使用阈值化操作，以保证处理的图像为二值图像
im = array(Image.open('houses.png').convert('L'))
im = 1*(im<128)

labels, nbr_objects = measurements.label(im)
print "Number of objects:", nbr_objects
```

上面的脚本首先载入该图像，通过阈值化方式来确保该图像是二值图像。通过和 1 相乘，脚本将布尔数组转换成二进制表示。然后，我们使用 label() 函数寻找单个的物体，并且按照它们属于哪个对象将整数标签给像素赋值。图 1-12b 是 labels 数组的图像。图像的灰度值表示对象的标签。可以看到，在一些对象之间有一些小的连接。进行二进制开（binary open）操作，我们可以将其移除：

```
# 形态学开操作更好地分离各个对象
im_open = morphology.binary_opening(im,ones((9,5)),iterations=2)

labels_open, nbr_objects_open = measurements.label(im_open)
print "Number of objects:", nbr_objects_open
```

binary_opening() 函数的第二个参数指定一个数组*结构元素*。该数组表示以一个像素为中心时，使用哪些相邻像素。在这种情况下，我们在 y 方向上使用 9 个像素（上面 4 个像素、像素本身、下面 4 个像素），在 x 方向上使用 5 个像素。你可以指定任意数组为结构元素，数组中的非零元素决定使用哪些相邻像素。参数 iterations 决定执行该操作的次数。你可以尝试使用不同的迭代次数 iterations 值，看一下对象的数目如何变化。你可以在图 1-12c 与图 1-12d 中查看经过开操作后的

注 1：这个图像实际上是图像“分割”后的结果。如果你想知道该图像是如何创建的，可以查看 9.3 节。

图像，以及相应的标签图像。正如你想象的一样，`binary_closing()` 函数实现相反的操作。我们将该函数和在 `morphology` 和 `measurements` 模块中的其他函数的用法留作练习。你可以从 `scipy.ndimage` 模块文档 http://docs.scipy.org/doc/scipy/reference/ndimage.html 中了解关于这些函数的更多知识。

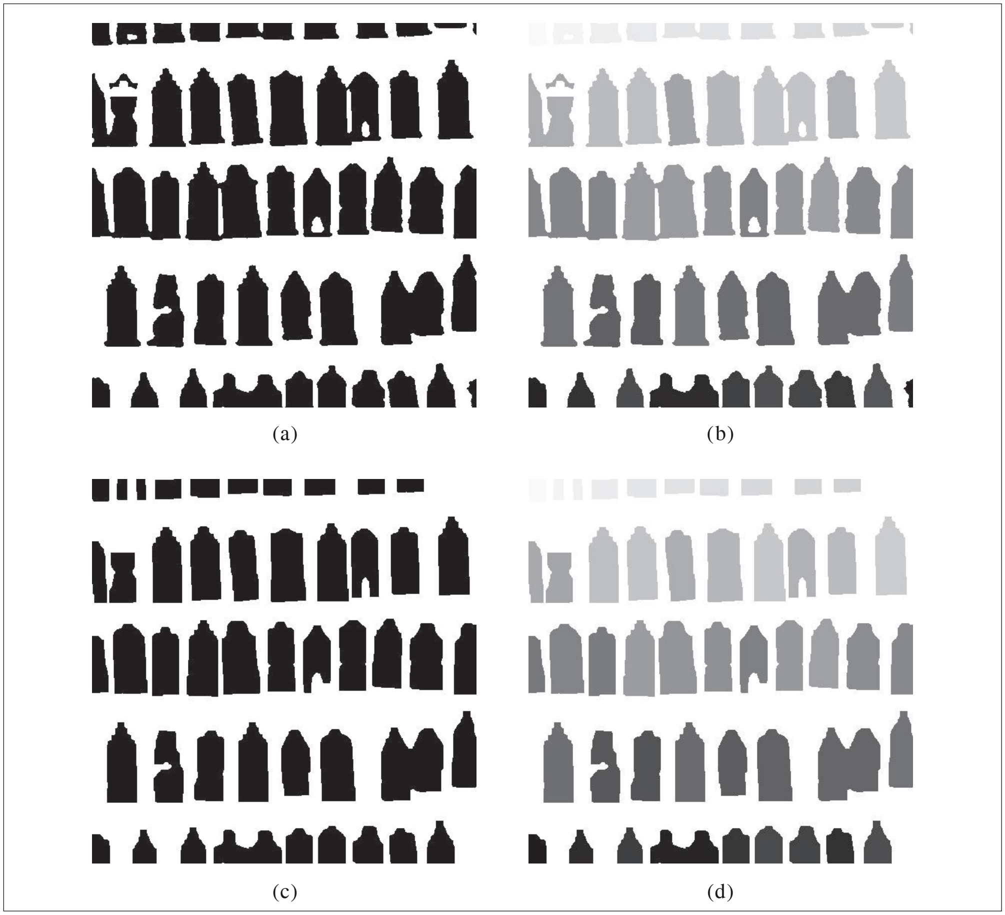


图 1-12：形态学示例。使用二值开操作将对象分开，然后计算物体的数目：（a）为原始二值图像；（b）为对应原始图像的标签图像，其中灰度值表示物体的标签；（c）为使用开操作后的二值图像；（d）为开操作后图像的标签图像

1.4.4 一些有用的SciPy模块

SciPy 中包含一些用于输入和输出的实用模块。下面介绍其中两个模块：`io` 和 `misc`。

1. 读写.mat文件

如果你有一些数据，或者在网上下载到一些有趣的数据集，这些数据以 Matlab

的 .mat 文件格式存储，那么可以使用 scipy.io 模块进行读取。

```
data = scipy.io.loadmat('test.mat')
```

上面代码中，data 对象包含一个字典，字典中的键对应于保存在原始 .mat 文件中的变量名。由于这些变量是数组格式的，因此可以很方便地保存到 .mat 文件中。你仅需创建一个字典（其中要包含你想要保存的所有变量），然后使用 savemat() 函数：

```
data = {}
data['x'] = x
scipy.io.savemat('test.mat',data)
```

因为上面的脚本保存的是数组 x，所以当读入到 Matlab 中时，变量的名字仍为 x。关于 scipy.io 模块的更多内容，请参见在线文档 http://docs.scipy.org/doc/scipy/reference/io.html。

2. 以图像形式保存数组

因为我们需要对图像进行操作，并且需要使用数组对象来做运算，所以将数组直接保存为图像文件 [1] 非常有用。本书中的很多图像都是这样的创建的。

imsave() 函数可以从 scipy.misc 模块中载入。要将数组 im 保存到文件中，可以使用下面的命令：

```
from scipy.misc import imsave
imsave('test.jpg',im)
```

scipy.misc 模块同样包含了著名的 Lena 测试图像：

```
lena = scipy.misc.lena()
```

该脚本返回一个 512×512 的灰度图像数组。

1.5 高级示例：图像去噪

我们通过一个非常实用的例子——图像的去噪——来结束本章。图像去噪是在去除图像噪声的同时，尽可能地保留图像细节和结构的处理技术。我们这里使用 ROF（Rudin-Osher-Fatemi）去噪模型。该模型最早出现在文献 [28] 中。图像去噪对于很多应用来说都非常重要；这些应用范围很广，小到让你的假期照片看起来更漂亮，大到提高卫星图像的质量。ROF 模型具有很好的性质：使处理后的图像更平滑，同时保持图像边缘和结构信息。

注 1：所有 Pylab 图均可保存为多种图像格式，方法是点击图像窗口中的“保存”按钮。

ROF 模型的数学基础和处理技巧非常高深，不在本书讲述范围之内。在讲述如何基于 Chambolle 提出的算法 [5] 实现 ROF 求解器之前，本书首先简要介绍一下 ROF 模型。

一幅（灰度）图像 $\boldsymbol{I}$ 的全变差（Total Variation，TV）定义为梯度范数之和。在连续表示的情况下，全变差表示为：

$$J(\boldsymbol{I}) = \int |\nabla \boldsymbol{I}| \mathrm{d}\mathbf{x} \tag{1.1}$$

在离散表示的情况下，全变差表示为：

$$J(\boldsymbol{I}) = \sum_{\mathbf{x}} |\nabla \boldsymbol{I}|$$

其中，上面的式子是在所有图像坐标 $\mathbf{x}=[x, y]$ 上取和。

在 Chambolle 提出的 ROF 模型里，目标函数为寻找降噪后的图像 $\boldsymbol{U}$，使下式最小：

$$\min_{U} ||\boldsymbol{I}-\boldsymbol{U}||^2 + 2\lambda J(\boldsymbol{U}),$$

其中范数 $||\boldsymbol{I}-\boldsymbol{U}||$ 是去噪后图像 $\boldsymbol{U}$ 和原始图像 $\boldsymbol{I}$ 差异的度量。也就是说，本质上该模型使去噪后的图像像素值“平坦”变化，但是在图像区域的边缘上，允许去噪后的图像像素值“跳跃”变化。

按照论文 [5] 中的算法，我们可以按照下面的代码实现 ROF 模型去噪：

```
from numpy import *

def denoise(im,U_init,tolerance=0.1,tau=0.125,tv_weight=100):
  """ 使用 A. Chambolle（2005）在公式（11）中的计算步骤实现 Rudin-Osher-Fatemi（ROF）去噪模型

    输入：含有噪声的输入图像（灰度图像）、U 的初始值、TV 正则项权值、步长、停业条件

    输出：去噪和去除纹理后的图像、纹理残留 """

  m,n = im.shape # 噪声图像的大小

  # 初始化
  U = U_init
  Px = im # 对偶域的 x 分量
  Py = im # 对偶域的 y 分量
  error = 1

while (error > tolerance):
  Uold = U
```

```
    # 原始变量的梯度
    GradUx = roll(U,-1,axis=1)-U # 变量 U 梯度的 x 分量
    GradUy = roll(U,-1,axis=0)-U # 变量 U 梯度的 y 分量

    # 更新对偶变量
    PxNew = Px + (tau/tv_weight)*GradUx
    PyNew = Py + (tau/tv_weight)*GradUy
    NormNew = maximum(1,sqrt(PxNew**2+PyNew**2))

    Px = PxNew/NormNew # 更新 x 分量（对偶）
    Py = PyNew/NormNew # 更新 y 分量（对偶）

    # 更新原始变量
    RxPx = roll(Px,1,axis=1) # 对 x 分量进行向右 x 轴平移
    RyPy = roll(Py,1,axis=0) # 对 y 分量进行向右 y 轴平移

    DivP = (Px-RxPx)+(Py-RyPy) # 对偶域的散度
    U = im + tv_weight*DivP # 更新原始变量

    # 更新误差
    error = linalg.norm(U-Uold)/sqrt(n*m);

  return U,im-U # 去噪后的图像和纹理残余
```

在这个例子中，我们使用了 `roll()` 函数。顾名思义，在一个坐标轴上，它循环“滚动”数组中的元素值。该函数可以非常方便地计算邻域元素的差异，比如这里的导数。我们还使用了 `linalg.norm()` 函数，该函数可以衡量两个数组间（这个例子中是指图像矩阵 ***U*** 和 *Uold*）的差异。我们将这个 `denoise()` 函数保存到 rof.py 文件中。

下面使用一个合成的噪声图像示例来说明如何使用该函数：

```
from numpy import *
from numpy import random
from scipy.ndimage import filters
import rof

# 使用噪声创建合成图像
im = zeros((500,500))
im[100:400,100:400] = 128
im[200:300,200:300] = 255
im = im + 30*random.standard_normal((500,500))

U,T = rof.denoise(im,im)
G = filters.gaussian_filter(im,10)

# 保存生成结果
from scipy.misc import imsave
imsave('synth_rof.pdf',U)
imsave('synth_gaussian.pdf',G)
```

原始图像和图像的去噪结果如图 1-13 所示。正如你所看到的，ROF 算法去噪后的图像很好地保留了图像的边缘信息。



(a) (b) (c)

图 1-13：使用 ROF 模型对合成图像去噪：（a）为原始噪声图像；（b）为经过高斯模糊的图像（σ=10）；（c）为经过 ROF 模型去噪后的图像

下面看一下在实际图像中使用 ROF 模型去噪的效果：

```
from PIL import Image
from pylab import *
import rof

im = array(Image.open('empire.jpg').convert('L'))
U,T = rof.denoise(im,im)

figure()
gray()
imshow(U)
axis('equal')
axis('off')
show()
```

经过 ROF 去噪后的图像如图 1-14c 所示。为了方便比较，该图中同样显示了模糊后的图像。可以看到，ROF 去噪后的图像保留了边缘和图像的结构信息，同时模糊了“噪声”。



图 1-14：使用 ROF 模型对灰度图像去噪：（a）为原始噪声图像；（b）为经过高斯模糊的图像（σ=5）；（c）为经过 ROF 模型去噪后的图像

练习

(1) 如图 1-9 所示，将一幅图像进行高斯模糊处理。随着 σ 的增加，绘制出图像轮廓。在你绘制出的图中，图像的轮廓有何变化？你能解释为什么会发生这些变化吗？

(2) 通过将图像模糊化，然后从原始图像中减去模糊图像，来实现*反锐化图像掩模*操作（http://en.wikipedia.org/wiki/Unsharp_masking）。反锐化图像掩模操作可以实现图像锐化效果。试在彩色和灰度图像上使用反锐化图像掩模操作，观察该操作的效果。

(3) 除了直方图均衡化，商图像是另一种图像归一化的方法。商图像可以通过除以模糊后的图像 $\boldsymbol{I}/(\boldsymbol{I} * \boldsymbol{G}_\sigma)$ 获得。尝试使用该方法，并使用一些样本图像进行验证。

(4) 使用图像梯度，编写一个在图像中获得简单物体（例如，白色背景中的正方形）轮廓的函数。

(5) 使用梯度方向和大小检测图像中的线段。估计线段的长度以及线段的参数，并在原始图像中重新绘制该线段。

(6) 使用 `label()` 函数处理二值化图像，并使用直方图和标签图像绘制图像中物体的大小分布。

(7) 使用形态学操作处理阈值化图像。在发现一些参数能够产生好的结果后，使用 `morphology` 模块里面的 `center_of_mass()` 函数寻找每个物体的中心坐标，将其在图像中绘制出来。

代码示例约定

从第 2 章起，我们假定 PIL、NumPy 和 Matplotlib 都包括在你所创建的每个文件和每个代码例子的开头：

```
from PIL import Image
from numpy import *
from pylab import *
```

这种约定使得示例代码更清晰，同时也便于读者理解。除此之外，我们使用 SciPy 模块时，将会在代码示例中显式声明。

一些纯化论者会反对这种将全体模块导入的方式，坚持如下使用方式：

```
import numpy as np
import matplotlib.pyplot as plt
```

这种方式能够保持命名空间（知道每个函数从哪儿来）。因为我们不需要 PyLab 中的 NumPy 部分，所以该例子只从 Matplotlib 中导入 pyplot 部分。纯化论者和经验丰富的程序员们知道这些区别，他们能够选择自己喜欢的方式。但是，为了使本书的内容和例子更容易被读者接受，我们不打算这样做。

请读者注意。

第2章

局部图像描述子

本章旨在寻找图像间的对应点和对应区域。本章将介绍用于图像匹配的两种局部描述子算法。本书的很多内容中都会用到这些局部特征，它们在很多应用中都有重要作用，比如创建全景图、增强现实技术以及计算图像的三维重建。

2.1 Harris角点检测器

Harris 角点检测算法（也称 Harris & Stephens 角点检测器）是一个极为简单的角点检测算法。该算法的主要思想是，如果像素周围显示存在多于一个方向的边，我们认为该点为兴趣点。该点就称为角点。

我们把图像域中点 **x** 上的对称半正定矩阵 $\boldsymbol{M}_I$=$\boldsymbol{M}_I$（**x**）定义为：

$$\boldsymbol{M}_I = \nabla \boldsymbol{I}\ \nabla \boldsymbol{I}^T = \begin{bmatrix} \boldsymbol{I}_x \\ \boldsymbol{I}_y \end{bmatrix} [\boldsymbol{I}_x \quad \boldsymbol{I}_y] = \begin{bmatrix} \boldsymbol{I}_x^{\ 2} & \boldsymbol{I}_x \boldsymbol{I}_y \\ \boldsymbol{I}_x \boldsymbol{I}_y & \boldsymbol{I}_y^2 \end{bmatrix} \tag{2.1}$$

其中 $\nabla \boldsymbol{I}$ 为包含导数 $\boldsymbol{I}_x$ 和 $\boldsymbol{I}_y$ 的图像梯度（我们已经在第 1 章定义了图像的导数和梯度）。由于该定义，$\boldsymbol{M}_I$ 的秩为 1，特征值为 λ_1=$|\nabla \boldsymbol{I}|^2$ 和 λ_2=0。现在对于图像的每一个像素，我们可以计算出该矩阵。

选择权重矩阵 $\boldsymbol{W}$（通常为高斯滤波器 G_σ），我们可以得到卷积：

$$\overline{\boldsymbol{M}}_I = \boldsymbol{W} * \boldsymbol{M}_I \tag{2.2}$$

该卷积的目的是得到 $\boldsymbol{M}_I$ 在周围像素上的局部平均。计算出的矩阵 $\overline{\boldsymbol{M}}_I$ 有称为 Harris

矩阵。$\boldsymbol{W}$ 的宽度决定了在像素 $\mathbf{x}$ 周围的感兴趣区域。像这样在区域附近对矩阵 $\overline{\boldsymbol{M}_I}$ 取平均的原因是，特征值会依赖于局部图像特性而变化。如果图像的梯度在该区域变化，那么 $\overline{\boldsymbol{M}_I}$ 的第二个特征值将不再为 0。如果图像的梯度没有变化，$\overline{\boldsymbol{M}_I}$ 的特征值也不会变化。

取决于该区域 $\nabla \boldsymbol{I}$ 的值，Harris 矩阵 $\overline{\boldsymbol{M}_I}$ 的特征值有三种情况：

- 如果 λ_1 和 λ_2 都是很大的正数，则该 $\mathbf{x}$ 点为角点；
- 如果 λ_1 很大，$\lambda_2 \approx 0$，则该区域内存在一个边，该区域内的平均 $\boldsymbol{M}_I$ 的特征值不会变化太大；
- 如果 $\lambda_1 \approx \lambda_2 \approx 0$，该区域内为空。

在不需要实际计算特征值的情况下，为了把重要的情况和其他情况分开，Harris 和 Stephens 在文献 [12] 中引入了指示函数：

$$\det(\overline{\boldsymbol{M}_I}) - \kappa \operatorname{trace}(\overline{\boldsymbol{M}_I})^2$$

为了去除加权常数 κ，我们通常使用商数：

$$\frac{\det(\overline{\boldsymbol{M}_I})}{\operatorname{trace}(\overline{\boldsymbol{M}_I})^2}$$

作为指示器。

下面我们写出 Harris 角点检测程序。像 1.4.2 节介绍的一样，对于这个函数，我们需要使用 `scipy.ndimage.filters` 模块中的高斯导数滤波器来计算导数。使用高斯滤波器的道理同样是，我们需要在角点检测过程中抑制噪声强度。

首先，将角点响应函数添加到 harris.py 文件中，该函数使用高斯导数实现。同样地，参数 σ 定义了使用的高斯滤波器的尺度大小。你也可以修改这个函数，对 x 和 y 方向上不同的尺度参数，以及尝试平均操作中的不同尺度，来计算 Harris 矩阵。

```
from scipy.ndimage import filters
def compute_harris_response(im,sigma=3):
  """ 在一幅灰度图像中，对每个像素计算 Harris 角点检测器响应函数 """

  # 计算导数
  imx = zeros(im.shape)
  filters.gaussian_filter(im, (sigma,sigma), (0,1), imx)
  imy = zeros(im.shape)
  filters.gaussian_filter(im, (sigma,sigma), (1,0), imy)
```

```
    # 计算 Harris 矩阵的分量
    Wxx = filters.gaussian_filter(imx*imx,sigma)
    Wxy = filters.gaussian_filter(imx*imy,sigma)
    Wyy = filters.gaussian_filter(imy*imy,sigma)

    # 计算特征值和迹
    Wdet = Wxx*Wyy - Wxy**2
    Wtr = Wxx + Wyy

    return Wdet / Wtr
```

上面的函数会返回像素值为 Harris 响应函数值的一幅图像。现在，我们需要从这幅图像中挑选出需要的信息。然后，选取像素值高于阈值的所有图像点；再加上额外的限制，即角点之间的间隔必须大于设定的最小距离。这种方法会产生很好的角点检测结果。为了实现该算法，我们获取所有的候选像素点，以角点响应值递减的顺序排序，然后将距离已标记为角点位置过近的区域从候选像素点中删除。将下面的函数添加到 harris.py 文件中：

```
def get_harris_points(harrisim,min_dist=10,threshold=0.1):
  """ 从一幅 Harris 响应图像中返回角点。min_dist 为分割角点和图像边界的最少像素数目 """

  # 寻找高于阈值的候选角点
  corner_threshold = harrisim.max() * threshold
  harrisim_t = (harrisim > corner_threshold) * 1

  # 得到候选点的坐标
  coords = array(harrisim_t.nonzero()).T

  # 以及它们的 Harris 响应值
  candidate_values = [harrisim[c[0],c[1]] for c in coords]

  # 对候选点按照 Harris 响应值进行排序
  index = argsort(candidate_values)

  # 将可行点的位置保存到数组中
  allowed_locations = zeros(harrisim.shape)
  allowed_locations[min_dist:-min_dist,min_dist:-min_dist] = 1

  # 按照 min_distance 原则，选择最佳 Harris 点
  filtered_coords = []
  for i in index:
    if allowed_locations[coords[i,0],coords[i,1]] == 1:
      filtered_coords.append(coords[i])
      allowed_locations[(coords[i,0]-min_dist):(coords[i,0]+min_dist),
```

```
            (coords[i,1]-min_dist):(coords[i,1]+min_dist)] = 0

    return filtered_coords
```

现在你有了检测图像中角点所需要的所有函数。为了显示图像中的角点，你可以使用 Matplotlib 模块绘制函数，将其添加到 harris.py 文件中，如下：

```
def plot_harris_points(image,filtered_coords):
  """ 绘制图像中检测到的角点 """

  figure()
  gray()
  imshow(image)
  plot([p[1] for p in filtered_coords],[p[0] for p in filtered_coords],'*')
  axis('off')
  show()
```

试着运行下面的命令：

```
im = array(Image.open('empire.jpg').convert('L'))
harrisim = harris.compute_harris_response(im)
filtered_coords = harris.get_harris_points(harrisim,6)
harris.plot_harris_points(im, filtered_coords)
```

首先，打开该图像，转换成灰度图像。然后，计算响应函数，基于响应值选择角点。最后，在原始图像中覆盖绘制检测出的角点。绘制出的结果图像如图 2-1 所示。

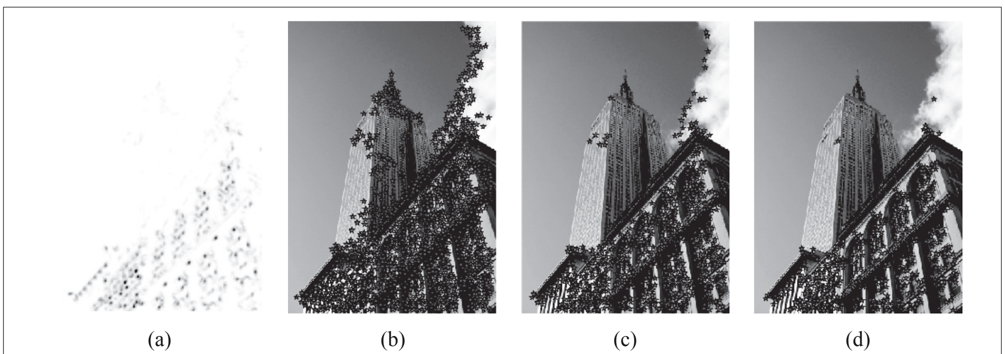


图 2-1：使用 Harris 角点检测器检测角点：（a）为 Harris 响应函数；（b-d）分别为使用阈值 0.01、0.05 和 0.1 检测出的角点

如果你想概要了解角点检测的不同方法，包括 Harris 角点检测器的改进和进一步的开发应用，可以查找资源，如网站 http://en.wikipedia.org/wiki/Corner_detection。

在图像间寻找对应点

Harris 角点检测器仅仅能够检测出图像中的兴趣点，但是没有给出通过比较图像间的兴趣点来寻找匹配角点的方法。我们需要在每个点上加入描述子信息，并给出一个比较这些描述子的方法。

兴趣点描述子是分配给兴趣点的一个向量，描述该点附近的图像的表观信息。描述子越好，寻找到的对应点越好。我们用对应点或者点的对应来描述相同物体和场景点在不同图像上形成的像素点。

Harris 角点的描述子通常是由周围图像像素块的灰度值，以及用于比较的归一化互相关矩阵构成的。图像的像素块由以该像素点为中心的周围矩形部分图像构成。

通常，两个（相同大小）像素块 $\boldsymbol{I}_1(\mathbf{x})$ 和 $\boldsymbol{I}_2(\mathbf{x})$ 的相关矩阵定义为：

$$c(\boldsymbol{I}_1,\boldsymbol{I}_2)=\sum_{\mathbf{x}} f(\boldsymbol{I}_1(\mathbf{x}),\boldsymbol{I}_2(\mathbf{x}))$$

其中，函数 f 随着相关方法的变化而变化。上式取像素块中所有像素位置 $\mathbf{x}$ 的和。对于互相关矩阵，函数 $f(\boldsymbol{I}_1，\boldsymbol{I}_2)=\boldsymbol{I}_1\boldsymbol{I}_2$, 因此，$c(\boldsymbol{I}_1，\boldsymbol{I}_2)=\boldsymbol{I}_1\cdot\boldsymbol{I}_2$, 其中 · 表示向量乘法（按照行或者列堆积的像素）。$c(\boldsymbol{I}_1，\boldsymbol{I}_2)$ 的值越大，像素块 $\boldsymbol{I}_1$ 和 $\boldsymbol{I}_2$ 的相似度越高。[1]

归一化的互相关矩阵是互相关矩阵的一种变形，可以定义为：

$$ncc(\boldsymbol{I}_1,\boldsymbol{I}_2)=\frac{1}{n-1}\sum_{\mathbf{x}}\frac{(\boldsymbol{I}_1(\mathbf{x})-\mu_1)}{\sigma_1}\cdot\frac{(\boldsymbol{I}_2(\mathbf{x})-\mu_2)}{\sigma_2} \qquad (2.3)$$

其中，n 为像素块中像素的数目，μ_1 和 μ_2 表示每个像素块中的平均像素值强度，σ_1 和 σ_2 分别表示每个像素块中的标准差。通过减去均值和除以标准差，该方法对图像亮度变化具有稳健性。

为获取图像像素块，并使用归一化的互相关矩阵来比较它们，你需要另外两个函数。将它们添加到 harris.py 文件中：

```
def get_descriptors(image,filtered_coords,wid=5):
  """ 对于每个返回的点，返回点周围 2*wid+1 个像素的值（假设选取点的 min_distance > wid）"""

  desc = []
  for coords in filtered_coords:
    patch = image[coords[0]-wid:coords[0]+wid+1,
```

注 1：另一个常用的函数是 $f(\boldsymbol{I}_1,\boldsymbol{I}_2)=(\boldsymbol{I}_1-\boldsymbol{I}_2)^2$，该函数表示平方差的和（Sum of Squared Difference，SSD）。

```
                coords[1]-wid:coords[1]+wid+1].flatten()
      desc.append(patch)

    return desc

def match(desc1,desc2,threshold=0.5):
    """ 对于第一幅图像中的每个角点描述子，使用归一化互相关，选取它在第二幅图像中的匹配角点 """

    n = len(desc1[0])

    # 点对的距离
    d = -ones((len(desc1),len(desc2)))
    for i in range(len(desc1)):
      for j in range(len(desc2)):
        d1 = (desc1[i] - mean(desc1[i])) / std(desc1[i])
        d2 = (desc2[j] - mean(desc2[j])) / std(desc2[j])
        ncc_value = sum(d1 * d2) / (n-1)
        if ncc_value > threshold:
          d[i,j] = ncc_value

    ndx = argsort(-d)
    matchscores = ndx[:,0]

    return matchscores
```

第一个函数的参数为奇数大小长度的方形灰度图像块，该图像块的中心为处理的像素点。该函数将图像块像素值压平成一个向量，然后添加到描述子列表中。第二个函数使用归一化的互相关矩阵，将每个描述子匹配到另一个图像中的最优的候选点。由于数值较高的距离代表两个点能够更好地匹配，所以在排序之前，我们对距离取相反数。为了获得更稳定的匹配，我们从第二幅图像向第一幅图像匹配，然后过滤掉在两种方法中不都是最好的匹配。下面的函数可以实现该操作：

```
def match_twosided(desc1,desc2,threshold=0.5):
    """ 两边对称版本的 match()"""

    matches_12 = match(desc1,desc2,threshold)
    matches_21 = match(desc2,desc1,threshold)

    ndx_12 = where(matches_12 >= 0)[0]

    # 去除非对称的匹配
    for n in ndx_12:
      if matches_21[matches_12[n]] != n:
        matches_12[n] = -1
```

```
    return matches_12
```

这些匹配可以通过在两边分别绘制出图像，使用线段连接匹配的像素点来直观地可视化。下面的代码可以实现匹配点的可视化。将这两个函数添加到 harris.py 文件中：

```
def appendimages(im1,im2):
  """ 返回将两幅图像并排拼接成的一幅新图像 """

  # 选取具有最少行数的图像，然后填充足够的空行
  rows1 = im1.shape[0]
  rows2 = im2.shape[0]

  if rows1 < rows2:
    im1 = concatenate((im1,zeros((rows2-rows1,im1.shape[1]))),axis=0)
  elif rows1 > rows2:
    im2 = concatenate((im2,zeros((rows1-rows2,im2.shape[1]))),axis=0)
  # 如果这些情况都没有，那么它们的行数相同，不需要进行填充

  return concatenate((im1,im2), axis=1)


def plot_matches(im1,im2,locs1,locs2,matchscores,show_below=True):
  """ 显示一幅带有连接匹配之间连线的图片
    输入：im1，im2（数组图像），locs1，locs2（特征位置），matchscores（match()的输出），
    show_below（如果图像应该显示在匹配的下方）"""

  im3 = appendimages(im1,im2)
  if show_below:
    im3 = vstack((im3,im3))

  imshow(im3)

  cols1 = im1.shape[1]
  for i,m in enumerate(matchscores):
    if m>0:
      plot([locs1[i][1],locs2[m][1]+cols1],[locs1[i][0],locs2[m][0]],'c')
  axis('off')
```

图 2-2 为使用归一化的互相关矩阵（在这个例子中，每个像素块的大小为 11 × 11）来寻找对应点的例子。该图像可以通过下面的命令实现：

```
wid = 5
harrisim = harris.compute_harris_response(im1,5)
filtered_coords1 = harris.get_harris_points(harrisim,wid+1)
```

```
d1 = harris.get_descriptors(im1,filtered_coords1,wid)

harrisim = harris.compute_harris_response(im2,5)
filtered_coords2 = harris.get_harris_points(harrisim,wid+1)
d2 = harris.get_descriptors(im2,filtered_coords2,wid)

print 'starting matching'
matches = harris.match_twosided(d1,d2)

figure()
gray()
harris.plot_matches(im1,im2,filtered_coords1,filtered_coords2,matches)
show()
```



图 2-2：将归一化的互相关矩阵应用于 Harris 角点周围图像块，来寻找匹配对应点

为了看得更清楚，你可以画出匹配的子集。在上面的代码中，可以通过将数组 matches 替换成 matches[:100] 或者任意子集来实现。

如图 2-2 所示，该算法的结果存在一些不正确匹配。这是因为，与现代的一些方法相比，图像像素块的互相关矩阵具有较弱的描述性。实际运用中，我们通常使用更稳健的方法来处理这些对应匹配。这些描述符还有一个问题，它们不具有尺度不变性和旋转不变性，而算法中像素块的大小也会影响对应匹配的结果。

近年来诞生了很多用来提高特征点检测和描述性能的方法。在下一节中，我们来学习其中最好的一种算法。

2.2 SIFT（尺度不变特征变换）

David Lowe 在文献 [17] 中提出的 SIFT（Scale-Invariant Feature Transform，尺度不变特征变换）是过去十年中最成功的图像局部描述子之一。SIFT 特征后来在文献 [18] 中得到精炼并详述，经受住了时间的考验。SIFT 特征包括兴趣点检测器和描述子。SIFT 描述子具有非常强的稳健性，这在很大程度上也是 SIFT 特征能够成功和流行的主要原因。自从 SIFT 特征的出现，许多其他本质上使用相同描述子的方法也相继出现。现在，SIFT 描述符经常和许多不同的兴趣点检测器相结合使用（有些情况下是区域检测器），有时甚至在整幅图像上密集地使用。SIFT 特征对于尺度、旋转和亮度都具有不变性，因此，它可以用于三维视角和噪声的可靠匹配。你可以在 http://en.wikipedia.org/wiki/Scale-invariant_feature_transform 获得 SIFT 特征的简要介绍。

2.2.1 兴趣点

SIFT 特征使用高斯差分函数来定位兴趣点：

$$D(\mathbf{x},\sigma)=[G_{\kappa\sigma}(\mathbf{x})-G_{\sigma}(\mathbf{x})]*\boldsymbol{I}(\mathbf{x})=[G_{\kappa\sigma}-G_{\sigma}]*\boldsymbol{I}=\boldsymbol{I}_{\kappa\sigma}-\boldsymbol{I}_{\sigma}$$

其中，G_σ 是上一章中介绍的二维高斯核，$\boldsymbol{I}_\sigma$ 是使用 G_σ 模糊的灰度图像，κ 是决定相差尺度的常数。兴趣点是在图像位置和尺度变化下 $D(\mathbf{x},\sigma)$ 的最大值和最小值点。这些候选位置点通过滤波去除不稳定点。基于一些准则，比如认为低对比度和位于边上的点不是兴趣点，我们可以去除一些候选兴趣点。你可以参考文献 [17, 18] 了解更多。

2.2.2 描述子

上面讨论的兴趣点（关键点）位置描述子给出了兴趣点的位置和尺度信息。为了实现旋转不变性，基于每个点周围图像梯度的方向和大小，SIFT 描述子又引入了参考

方向。SIFT 描述子使用主方向描述参考方向。主方向使用方向直方图（以大小为权重）来度量。

下面我们基于位置、尺度和方向信息来计算描述子。为了对图像亮度具有稳健性，SIFT 描述子使用图像梯度（之前 Harris 描述子使用图像亮度信息计算归一化互相关矩阵）。SIFT 描述子在每个像素点附近选取子区域网格，在每个子区域内计算图像梯度方向直方图。每个子区域的直方图拼接起来组成描述子向量。SIFT 描述子的标准设置使用 4×4 的子区域，每个子区域使用 8 个小区间的方向直方图，会产生共 128 个小区间的直方图（4×4×8=128）。图 2-3 所示为描述子的构造过程。感兴趣的读者可以参考文献 [18] 获取更多内容，或者从 http://en.wikipedia.org/wiki/Scale-invariant_feature_transform 概要了解 SIFT 特征描述子。

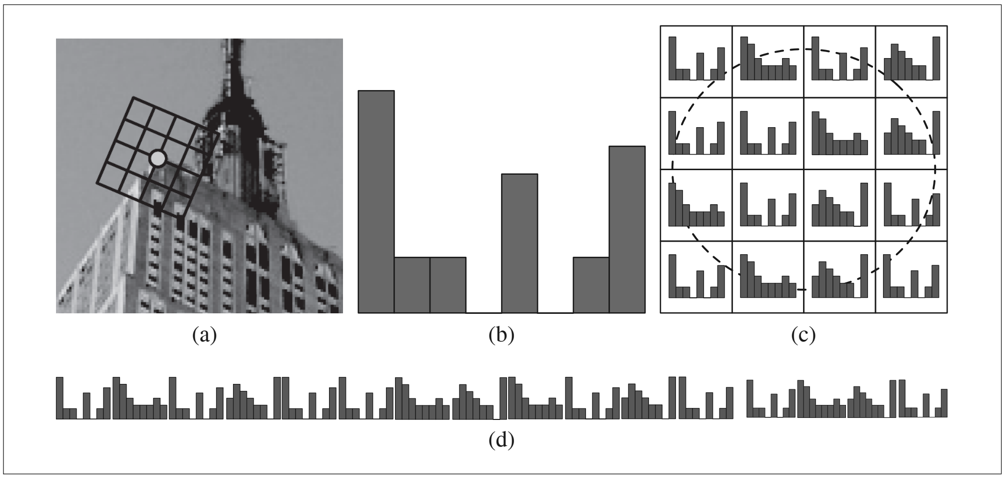


图 2-3：构造 SIFT 描述子特征向量的图解：（a）一个围绕兴趣点的网格结构，其中该网格已经按照梯度主方向进行了旋转；（b）在网格的一个子区域内构造梯度方向的 8-bin 直方图；（c）在网格的每个子区域内提取直方图；（d）拼接直方图，得到一个长的特征向量

2.2.3 检测兴趣点

我们使用开源工具包 VLFeat 提供的二进制文件来计算图像的 SIFT 特征 [36]。用完整的 Python 实现 SIFT 特征的所有步骤可能效率不是很高，并且超出了本书的范围。VLFeat 工具包可以从 http://www.vlfeat.org/ 下载，二进制文件可以在所有主要的平台上运行。VLFeat 库是用 C 语言来写的，但是我们可以使用该库提供的命令行接口。如果你认为使用 Matlab 接口或者 Python 包装器比二进制文件更方便，可以从 http://github.com/mmmikael/vlfeat/ 下载相应的版本。由于 Python 包装器对平台的依赖性，安装 Python 包装器在某些平台上需要一定的技巧，所以我们这里使用二进制

文件版本。Lowe 的个人网站上也有 SIFT 特征的实现，可以参见 http://www.cs.ubc.ca/~lowe/keypoints/，该代码仅适用于 Windows 系统和 Linux 系统。

创建 sift.py 文件，将下面调用可执行文件的函数添加到该文件中：

```
def process_image(imagename,resultname,params="--edge-thresh 10 --peak-thresh 5"):
  """ 处理一幅图像，然后将结果保存在文件中 """

  if imagename[-3:] != 'pgm':
    # 创建一个 pgm 文件
    im = Image.open(imagename).convert('L')
    im.save('tmp.pgm')
    imagename = 'tmp.pgm'

  cmmd = str("sift "+imagename+" --output="+resultname+
        " "+params)
  os.system(cmmd)
  print 'processed', imagename, 'to', resultname
```

由于该二进制文件需要的图像格式为灰度 .pgm，所以如果图像为其他格式，我们需要首先将其转换成 .pgm 格式文件。转换的结果以易读的格式保存在文本文件中。文本文件如下：

```
318.861 7.48227 1.12001 1.68523 0 0 0 1 0 0 0 0 0 11 16 0 ...
318.861 7.48227 1.12001 2.99965 11 2 0 0 1 0 0 0 173 67 0 0 ...
54.2821 14.8586 0.895827 4.29821 60 46 0 0 0 0 0 0 99 42 0 0 ...
155.714 23.0575 1.10741 1.54095 6 0 0 0 150 11 0 0 150 18 2 1 ...
42.9729 24.2012 0.969313 4.68892 90 29 0 0 0 1 2 10 79 45 5 11 ...
229.037 23.7603 0.921754 1.48754 3 0 0 0 141 31 0 0 141 45 0 0 ...
232.362 24.0091 1.0578 1.65089 11 1 0 16 134 0 0 0 106 21 16 33 ...
201.256 25.5857 1.04879 2.01664 10 4 1 8 14 2 1 9 88 13 0 0 ...
⋮
```

上面数据的每一行前 4 个数值依次表示兴趣点的坐标、尺度和方向角度，后面紧接着的是对应描述符的 128 维向量。这里的描述子使用原始整数数值表示，没有经过归一化处理。当你需要比较这些描述符时，要做一些处理。更多的内容请见后面的介绍。

上面的例子显示的是在一幅图像中前 8 个特征的前面部分数值。注意前两行的坐标值相同，但是方向不同。当同一个兴趣点上出现不同的显著方向，这种情况就会出现的。

下面是如何从像上面的输出文件中，将特征读取到 NumPy 数组中的函数。将该函数添加到 sift.py 文件中：

```
def read_features_from_file(filename):
  """ 读取特征属性值，然后将其以矩阵的形式返回 """

  f = loadtxt(filename)
  return f[:,:4],f[:,4:] # 特征位置，描述子
```

在上面的函数中，我们使用 NumPy 库中的 loadtxt() 函数来处理所有的工作。

如果在 Python 会话中修改描述子，你需要将输出结果保存到特征文件中。下面的函数使用 NumPy 库中的 savetxt() 函数，可以帮你实现该功能：

```
def write_features_to_file(filename,locs,desc):
  """ 将特征位置和描述子保存到文件中 """
  savetxt(filename,hstack((locs,desc)))
```

上面的函数使用了 hstack() 函数。该函数通过拼接不同的行向量来实现水平堆叠两个向量的功能。在这个例子中，每一行中前几列为位置信息，紧接着是描述子。

读取特征后，通过在图像上绘制出它们的位置，可以将其可视化。将下面的 plot_features() 函数添加到 sift.py 文件中，可以实现该功能。

```
def plot_features(im,locs,circle=False):
  """ 显示带有特征的图像
    输入：im（数组图像），locs（每个特征的行、列、尺度和方向角度）"""

  def draw_circle(c,r):
    t = arange(0,1.01,.01)*2*pi
    x = r*cos(t) + c[0]
    y = r*sin(t) + c[1]
    plot(x,y,'b',linewidth=2)

  imshow(im)
  if circle:
    for p in locs:
      draw_circle(p[:2],p[2])
  else:
    plot(locs[:,0],locs[:,1],'ob')
  axis('off')
```

该函数在原始图像上使用蓝色的圆圈绘制出 SIFT 特征点的位置。将参数 circle 的选项设置为 True，该函数将使用 draw_circle() 函数绘制出圆圈，圆圈的半径为特征的尺度。

你可以通过下面的命令绘制出如图 2-4b 中 SIFT 特征位置的图像：

```
import sift

imname = 'empire.jpg'
im1 = array(Image.open(imname).convert('L'))
sift.process_image(imname,'empire.sift')
l1,d1 = sift.read_features_from_file('empire.sift')

figure()
gray()
sift.plot_features(im1,l1,circle=True)
show()
```

为了比较 Harris 角点和 SIFT 特征的不同，右图（图 2-4c）显示的是同一幅图像的 Harris 角点。你可以看到，两个算法所选择特征点的位置不同。

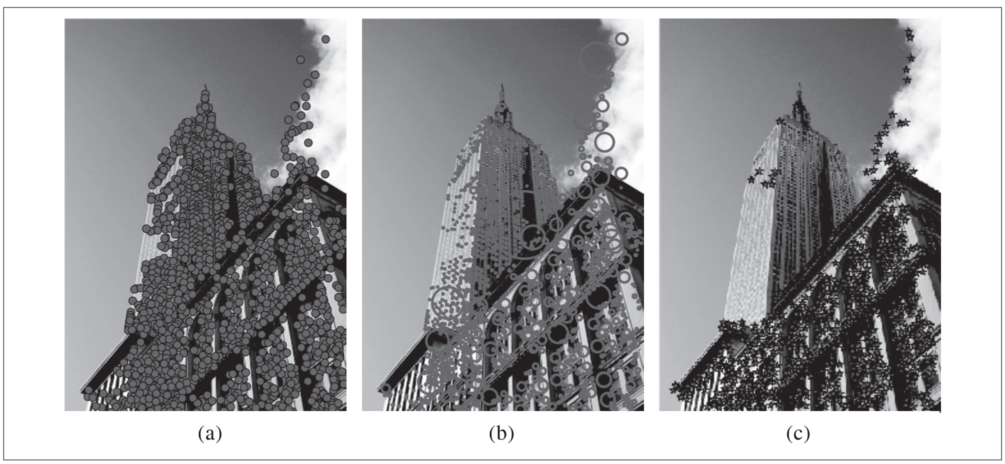


图 2-4：对一幅图像提取 SIFT 特征。（a）SIFT 特征；（b）使用圆圈表示特征尺度的 SIFT 特征；（c）为了比较，对于同一幅图像提取 Harris 角点

2.2.4 匹配描述子

对于将一幅图像中的特征匹配到另一幅图像的特征，一种稳健的准则（同样是由 Lowe 提出的）是使用这两个特征距离和两个最匹配特征距离的比率。相比于图像中的其他特征，该准则保证能够找到足够相似的唯一特征。使用该方法可以使错误的匹配数降低。下面的代码实现了匹配函数。将 match() 函数添加到 sift.py 文件中：

```
def match(desc1,desc2):
  """ 对于第一幅图像中的每个描述子，选取其在第二幅图像中的匹配
    输入：desc1（第一幅图像中的描述子），desc2（第二幅图像中的描述子）"""
```

```
desc1 = array([d/linalg.norm(d) for d in desc1])
desc2 = array([d/linalg.norm(d) for d in desc2])

dist_ratio = 0.6
desc1_size = desc1.shape

matchscores = zeros((desc1_size[0],1),'int')
desc2t = desc2.T                   # 预先计算矩阵转置
for i in range(desc1_size[0]):
  dotprods = dot(desc1[i,:],desc2t) # 向量点乘
  dotprods = 0.9999*dotprods
  # 反余弦和反排序，返回第二幅图像中特征的索引
  indx = argsort(arccos(dotprods))

  # 检查最近邻的角度是否小于 dist_ratio 乘以第二近邻的角度
  if arccos(dotprods)[indx[0]] < dist_ratio * arccos(dotprods)[indx[1]]:
  matchscores[i] = int(indx[0])

return matchscores
```

该函数使用描述子向量间的夹角作为距离度量。在此之前，我们需要将描述子向量归一化到单位长度[1]。因为这种匹配是单向的，即我们将每个特征向另一幅图像中的所有特征进行匹配，所以我们可以先计算第二幅图像兴趣点描述子向量的转置矩阵。这样，我们就不需要对每个特征分别进行转置操作。

为了进一步增加匹配的稳健性，我们可以再反过来执行一次该步骤，用另外的方法匹配（从第二幅图像中的特征向第一幅图像中的特征匹配）。最后，我们仅保留同时满足这两种匹配准则的对应（和我们对 Harris 角点的处理方法相同）。下面的 match_twosided() 函数可以实现该操作：

```
def match_twosided(desc1,desc2):
  """ 双向对称版本的 match()"""

  matches_12 = match(desc1,desc2)
  matches_21 = match(desc2,desc1)

  ndx_12 = matches_12.nonzero()[0]

  # 去除不对称的匹配
  for n in ndx_12:
    if matches_21[int(matches_12[n])] != n:
      matches_12[n] = 0

  return matches_12
```

注 1：对于单位向量，向量乘积（不使用 arccos() 函数）等价于标准欧式距离度量。

为了绘制出这些匹配点，我们可以使用在 harris.py 用到的相同函数。方便起见，将 `appendimages()` 函数和 `plot_matches()` 函数复制 过来。然后，将它们添加到 sift.py 文件中。如果你喜欢，也可以通过载入 harris.py 来使用这两个函数。

图 2-5 和图 2-6 是在图像对中检测 SIFT 特征点的例子，以及通过 `match_twosided()` 函数返回的特征点匹配情况。

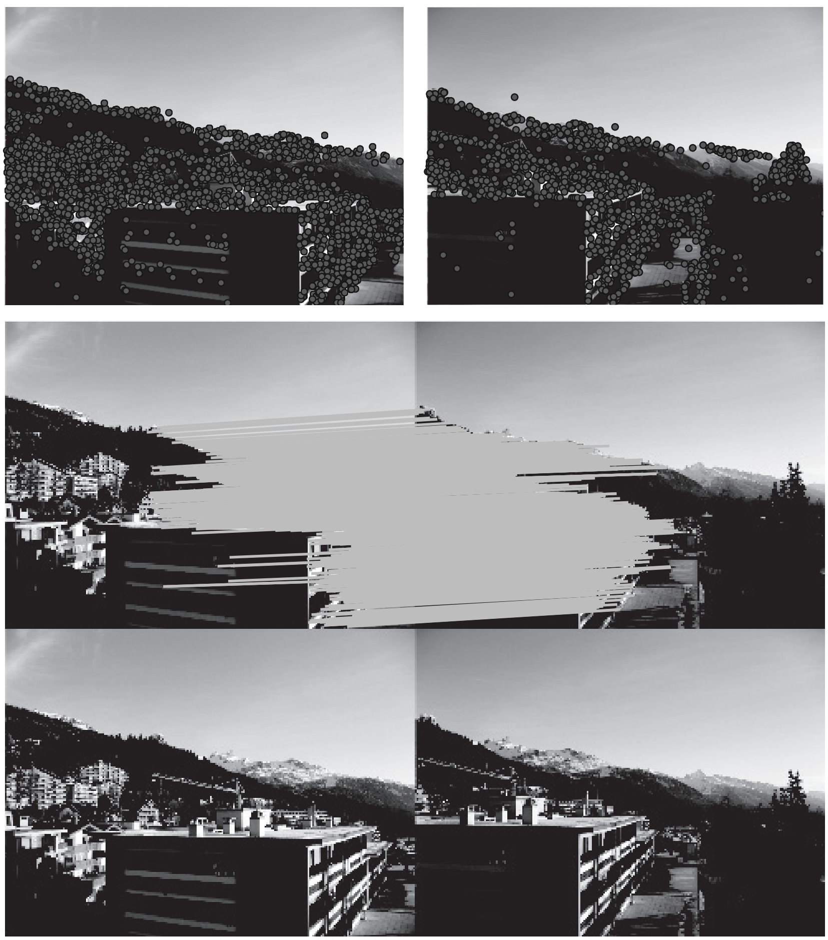
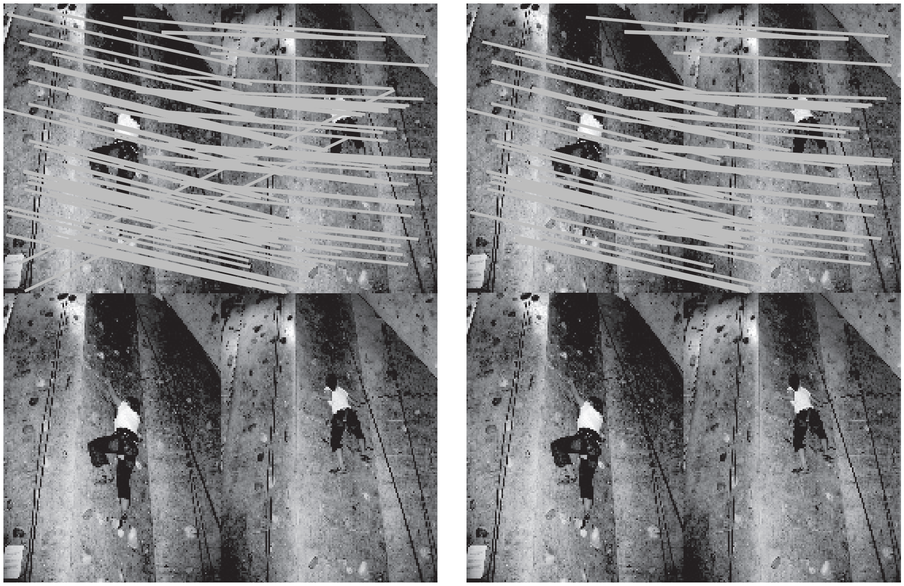


图 2-5：在两幅图像间检测和匹配 SIFT 特征



图 2-6：在两幅图像间检测和匹配 SIFT 特征

图 2-7 是在两幅图像中分别使用 `match()` 函数和 `match_twosided()` 函数匹配特征的另一个例子。正如你所看到的一样，使用对称（两边）匹配条件可以去除不正确的匹配，保留好的匹配（一些正确的匹配也被去除了）。



(a) (b)

图 2-7：在两幅图像间匹配 SIFT 特征的例子：（a）不使用两边匹配函数，将左边图像中的特征向右边图像中的特征匹配；（b）使用两边匹配函数后，剩余的匹配情况

通过检测和匹配特征点，我们可以将这些局部描述子应用到很多例子中。为了稳健地过滤掉这些不正确的匹配，接下来的两个章节将会在对应上加入几何学的约束关系，并将局部描述子应用到一些例子中，比如自动创建全景图、照相机姿态估计以及三维结构计算。

2.3 匹配地理标记图像

我们将通过一个示例应用来结束本章节。在这个例子中，我们使用局部描述子来匹配带有地理标记的图像。

2.3.1 从Panoramio下载地理标记图像

你可以从谷歌提供的照片共享服务 Panoramio（http://www.panoramio.com/）获得地理标记图像。像许多网络资源一样，Panoramio 提供一个 API 接口，方便用户使用程序访问这些内容。Panoramio 的 API 非常简单直接，可以在 http://www.panoramio.com/api/ 上找到 API 的使用方式。你可以通过 HTTP GET 方式访问网址内容，如下：

```
http://www.panoramio.com/map/get_panoramas.php?order=popularity&set=public&
from=0&to=20&minx=-180&miny=-90&maxx=180&maxy=90&size=medium
```

其中的 minx、miny、maxx 和 maxy 定义了选取照片的地理区域位置（分别表示最小经度、最小纬度、最大经度和最大纬度），你会得到可以简单解析的 JSON 格式的响应。JSON 是用于网络服务间数据传输的常用格式，比 XML 和其他格式更轻便。你可以从 http://en.wikipedia.org/wiki/JSON 获取更多关于 JSON 的内容。

你可以使用两个不同的视点来看华盛顿白宫的位置，通常从宾夕法尼亚大街南侧拍摄，或者从北侧拍摄。其坐标（纬度、经度）如下：

```
lt=38.897661
ln=-77.036564
```

为了转换成 API 调用需要的格式，需要在这些坐标值上减去或者加上一个数值，来获得以白宫为中心的正方形范围内的所有图像。调用如下：

```
http://www.panoramio.com/map/get_panoramas.php?order=popularity&set=public&
from=0&to=20&minx=-77.037564&miny=38.896662&maxx=-77.035564&maxy=38.898662&
size=medium
```

该调用返回在坐标边界内（±0.001）的前 20 幅图像，这些图像按照用户访问情况排序。调用的响应格式如下：


```
{ "count": 349,
"photos": [{"photo_id": 7715073, "photo_title": "White House", "photo_url":
"http://www.panoramio.com/photo/7715073", "photo_file_url":
"http://mw2.google.com/mw-panoramio/photos/medium/7715073.jpg", "longitude":
-77.036583, "latitude": 38.897488, "width": 500, "height": 375, "upload_date":
"10 February 2008", "owner_id": 1213603, "owner_name": "***", "owner_url":
"http://www.panoramio.com/user/1213603"}
,
{"photo_id": 1303971, "photo_title": "White House balcony", "photo_url":
"http://www.panoramio.com/photo/1303971", "photo_file_url":
"http://mw2.google.com/mw-panoramio/photos/medium/1303971.jpg", "longitude":
-77.036353, "latitude": 38.897471, "width": 500, "height": 336, "upload_date":
"13 March 2007", "owner_id": 195000, "owner_name": "***", "owner_url":
"http://www.panoramio.com/user/195000"}
...
]}
```


为了解析这个 JSON 格式的响应，我们可以使用 `simplejson` 工具包，可以从 http://github.com/simplejson/simplejson 下载。在项目界面上，可以看到在线的说明文档。

如果你使用的 Python 是 2.6 或之后的版本，因为在这些后来版本中已经包含 JSON 库，所以不需要使用 simplejson 工具包。如果想使用内置的 JSON 库，你只需要像这样导入即可：

```
import json
```

如果你想使用上面链接中的 simplejson 工具包（速度很快，并且比内置包含更新的内容），一个非常好的办法是使用可靠的方式导入它，如下：

```
try: import simplejson as json
except ImportError: import json
```

下面的代码将使用 Python 里的 urllib 工具包来处理请求，然后使用 simplejson 工具包解析返回结果：

```
import os
import urllib, urlparse
import simplejson as json

# 查询图像
url = 'http://www.panoramio.com/map/get_panoramas.php?order=popularity&\
    set=public&from=0&to=20&minx=-77.037564&miny=38.896662&\
    maxx=-77.035564&maxy=38.898662&size=medium'
c = urllib.urlopen(url)

# 从 JSON 中获得每个图像的 url
j = json.loads(c.read())
imurls = []
for im in j['photos']:
    imurls.append(im['photo_file_url'])

# 下载图像
for url in imurls:
    image = urllib.URLopener()
    image.retrieve(url, os.path.basename(urlparse.urlparse(url).path))
    print 'downloading:', url
```

通过 JSON 的输出可以看到，我们需要的是 photo_file_url 字段。运行上面的代码，在控制台上你应该能够看到类似下面的数据：

```
downloading: http://mw2.google.com/mw-panoramio/photos/medium/7715073.jpg
downloading: http://mw2.google.com/mw-panoramio/photos/medium/1303971.jpg
downloading: http://mw2.google.com/mw-panoramio/photos/medium/270077.jpg
downloading: http://mw2.google.com/mw-panoramio/photos/medium/15502.jpg
...
```

图 2-8 是本例子中返回的 20 幅图像。接下来，我们仅仅需要找到并匹配这些图像对之间的特征。



图 2-8：从 panoramio.com 下载的在同一个地理位置点（以白宫为中心的方形区域）上拍摄的图像

2.3.2 使用局部描述子匹配

我们刚才已经下载了这些图像，下面需要对这些图像提取局部描述子。在这种情况下，我们将使用前面部分讲述的 SIFT 特征描述子。我们假设已经对这些图像使用 SIFT 特征提取代码进行了处理，并且将特征保存在和图像同名（但文件名后缀是 .sift，而不是 .jpg）的文件中。假设 imlist 和 featlist 列表中包含这些文件名。我们可以对所有组合图像对进行逐个匹配，如下：

```
import sift

nbr_images = len(imlist)

matchscores = zeros((nbr_images,nbr_images))
for i in range(nbr_images):
  for j in range(i,nbr_images): # 仅仅计算上三角
```

```
    print 'comparing ', imlist[i], imlist[j]

    l1,d1 = sift.read_features_from_file(featlist[i])
    l2,d2 = sift.read_features_from_file(featlist[j])

    matches = sift.match_twosided(d1,d2)

    nbr_matches = sum(matches > 0)
    print 'number of matches = ', nbr_matches
    matchscores[i,j] = nbr_matches

# 复制值
for i in range(nbr_images):
  for j in range(i+1,nbr_images): # 不需要复制对角线
    matchscores[j,i] = matchscores[i,j]
```

我们将每对图像间的匹配特征数保存在 matchscores 数组中。因为该“距离度量”是对称的，所以我们可以不在代码的最后部分复制数值，来将 matchscores 矩阵填充完整；填充完整后的 matchscores 矩阵只是看起来更好。这些特定图像的 matchscores 矩阵里的数值如下：

```
662 0 0 2 0 0 0 0 1 0 0 1 2 0 3 0 19 1 0 2
0 901 0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 1 2
0 0 266 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
2 1 0 1481 0 0 2 2 0 0 0 2 2 0 0 0 2 3 2 0
0 0 0 0 1748 0 0 1 0 0 0 0 0 2 0 0 0 0 0 1
0 0 0 0 0 1747 0 0 1 0 0 0 0 0 0 0 0 1 1 0
0 0 0 2 0 0 555 0 0 0 1 4 4 0 2 0 0 5 1 0
0 1 0 2 1 0 0 2206 0 0 0 1 0 0 1 0 2 0 1 1
1 1 0 0 0 1 0 0 629 0 0 0 0 0 0 0 1 0 0 20
0 0 0 0 0 0 0 0 0 829 0 0 1 0 0 0 0 0 0 2
0 0 0 0 0 0 1 0 0 0 1025 0 0 0 0 0 1 1 1 0
1 1 0 2 0 0 4 1 0 0 0 528 5 2 15 0 3 6 0 0
2 0 0 2 0 0 4 0 0 1 0 5 736 1 4 0 3 37 1 0
0 0 1 0 2 0 0 0 0 0 0 2 1 620 1 0 0 1 0 0
3 0 0 0 0 0 2 1 0 0 0 15 4 1 553 0 6 9 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2273 0 1 0 0
19 0 0 2 0 0 0 2 1 0 1 3 3 0 6 0 542 0 0 0
1 0 0 3 0 1 5 0 0 0 1 6 37 1 9 1 0 527 3 0
0 1 0 2 0 1 1 1 0 0 1 0 1 0 1 0 0 3 1139 0
2 2 0 0 1 0 0 1 20 2 0 0 0 0 0 0 0 0 0 499
```

使用该 matchscores 矩阵作为图像间简单的距离度量方式（具有相似内容的图像间拥有更多的匹配特征数），下面我们可以使用相似的视觉内容来将这些图像连接起来。

2.3.3 可视化连接的图像

我们首先通过图像间是否具有匹配的局部描述子来定义图像间的连接，然后可视化这些连接情况。为了完成可视化，我们可以在图中显示这些图像，图的边代表连接。我们将会使用 pydot 工具包（http://code.google.com/p/pydot/），该工具包是功能强大的 GraphViz 图形库的 Python 接口。Pydot 使用 Pyparsing（http://pyparsing.wikispaces.com/）和 GraphViz（http://www.graphviz.org/）；不用担心，这些都非常容易安装，只需要几分钟就可以安装成功。

Pydot 非常容易使用。下面的一小段代码很好地展示了这一点。该代码会创建一个图，该图表示深度为 2 的树，具有 5 个分支，将分支的编号添加到分支节点上。图的结构如图 2-9 所示。我们有很多方法来修改图的布局和外观。如果你想了解更多内容，可以查看 Pydot 的说明文档，或者在 http://www.graphviz.org/Documentation.php 查看 GraphViz 使用的 DOT 语言介绍。

```
import pydot

g = pydot.Dot(graph_type='graph')

g.add_node(pydot.Node(str(0),fontcolor='transparent'))
for i in range(5):
  g.add_node(pydot.Node(str(i+1)))
  g.add_edge(pydot.Edge(str(0),str(i+1)))
  for j in range(5):
    g.add_node(pydot.Node(str(j+1)+'-'+str(i+1)))
    g.add_edge(pydot.Edge(str(j+1)+'-'+str(i+1),str(j+1)))
g.write_png('graph.jpg',prog='neato')
```

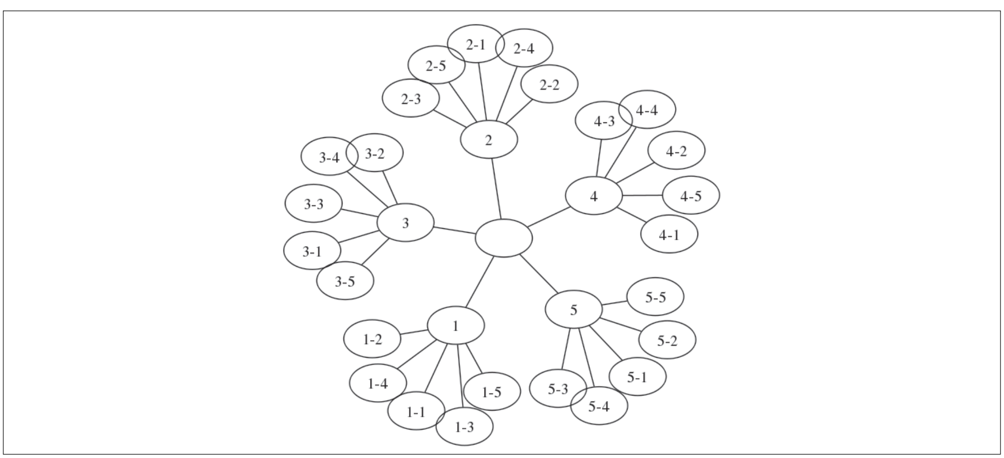


图 2-9：使用 pydot 工具包创建图

我们接下来继续探讨地理标记图像处理的例子。为了创建显示可能图像组的图，如果匹配的数目高于一个阈值，我们使用边来连接相应的图像节点。为了得到图中的图像，需要使用图像的全路径（在下面例子中，使用 *path* 变量表示）。为了使图像看起来漂亮，我们需要将每幅图像尺度化为缩略图形式，缩略图的最大边为 100 像素。下面是具体实现代码：

```
import pydot

threshold = 2 # 创建关联需要的最小匹配数目

g = pydot.Dot(graph_type='graph') # 不使用默认的有向图
for i in range(nbr_images):
  for j in range(i+1,nbr_images):
    if matchscores[i,j] > threshold:
        # 图像对中的第一幅图像
        im = Image.open(imlist[i])
        im.thumbnail((100,100))
        filename = str(i)+'.png'
        im.save(filename) # 需要一定大小的临时文件
        g.add_node(pydot.Node(str(i),fontcolor='transparent',
              shape='rectangle',image=path+filename))

        # 图像对中的第二幅图像
        im = Image.open(imlist[j])
        im.thumbnail((100,100))
        filename = str(j)+'.png'
        im.save(filename) # 需要一定大小的临时文件
        g.add_node(pydot.Node(str(j),fontcolor='transparent',
                shape='rectangle',image=path+filename))

        g.add_edge(pydot.Edge(str(i),str(j)))

  g.write_png('whitehouse.png')
```

代码运行结果如图 2-10 所示。图的具体内容和结构取决于你下载的图像。对于这个特定的例子，我们使用两组图像，每组分别是两个视角的白宫图像。

这个应用是使用局部描述子来匹配图像间区域的一个简单例子。在该应用中，我们没有使用针对任何匹配的限制约束。匹配的约束（具有很强的稳健性）可以通过接下来两章中的内容来实现。

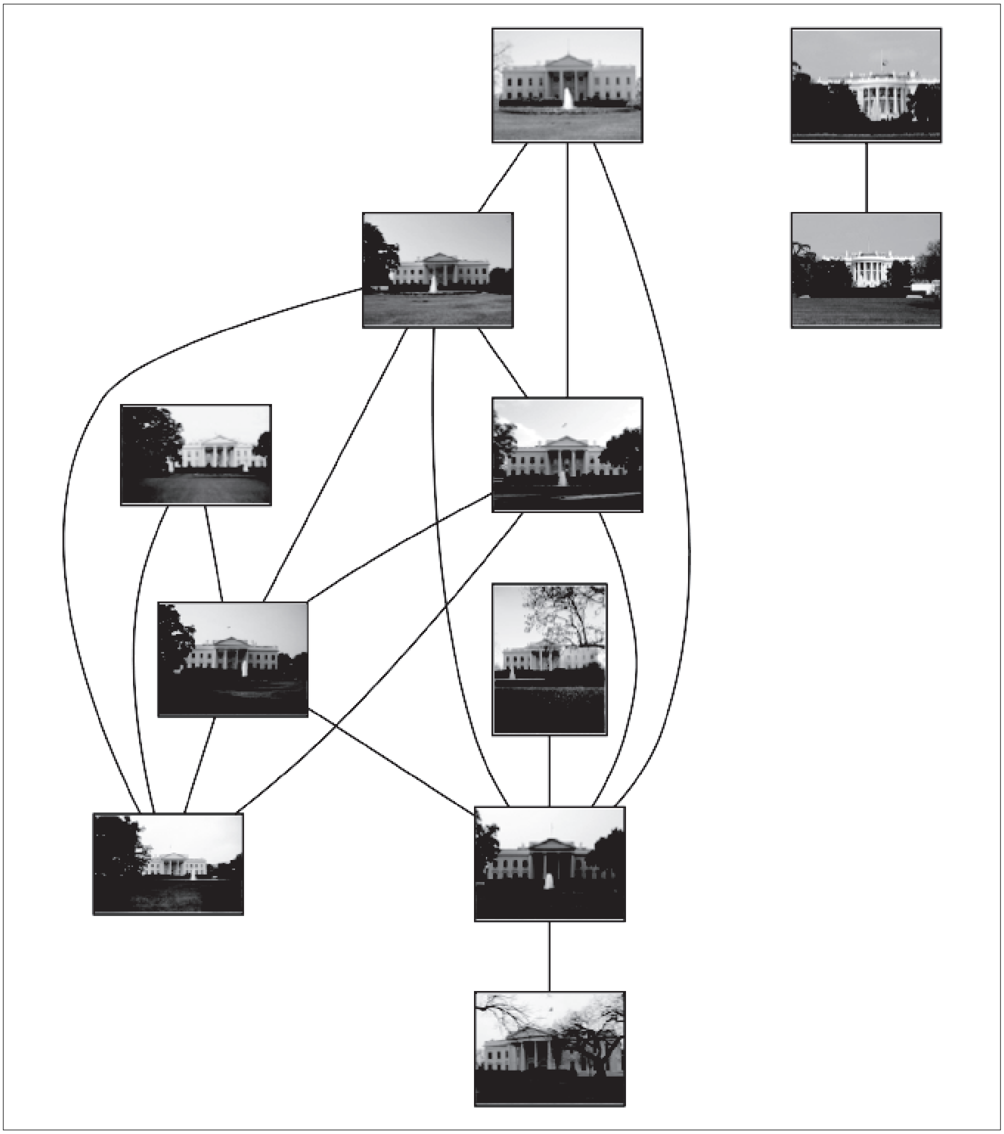


图 2-10：使用局部描述子将在同一地理位置点拍摄图像进行分类

练习

(1) 为了让匹配具有更强的稳健性，修改用于匹配 Harris 角点的函数，使其输入参数中包含认为两点存在对应关系允许的最大像素距离。

(2) 对一幅图像不断地应用模糊操作（或者 ROF 去噪），使得模糊效果越来越强，然后提取 Harris 角点，会出现什么问题？

(3) 另一种 Harris 角点检测器是快速角点检测器。有很多快速角点检测器的实现方法，包括纯 Python 语言实现的版本，可以在 http://www.edwardrosten.com/work/fast.html 下载。尝试使用该检测器，使用敏感性的阈值，然后将结果和 Harris 角点检测器检测出的角点比较。

(4) 以不同分辨率创建一幅图像的副本（例如，可以尝试多次将图像的尺寸减半）。对每幅图像提取 SIFT 特征。绘制以及匹配特征，来发现尺度的独立性是如何以及何时失效的。

(5) VLFeat 命令行工具同样实现了最大稳定极值区域（MSER，http://en.wikipedia.org/wiki/Maximally_stable_extremal_regions）算法。该算法是个能够找到角点一侧区域的区域检测器。创建一个用于提取 MSER 区域的函数，然后使用 `-read-frames` 选项将它们传递给 SIFT 特征描述子部分，最后写出一个用于绘制该区域边界的函数。

(6) 基于对应关系，写出在图像对间匹配特征的函数，以实现估计尺度差异以及场景的平面旋转。

(7) 任意选取一个位置，然后下载该位置的图像，像白宫例子一样将它们匹配起来。你能发现用于连接这些图像的更好方式吗？你是如何利用图来选取用于地理位置具有代表性的图像的？

第 3 章

图像到图像的映射

本章讲解图像之间的变换，以及一些计算变换的实用方法。这些变换可以用于图像扭曲变形和图像配准。最后，我们将会介绍一个自动创建全景图像的例子。

3.1 单应性变换

单应性变换是将一个平面内的点映射到另一个平面内的二维投影变换。在这里，平面是指图像或者三维中的平面表面。单应性变换具有很强的实用性，比如图像配准、图像纠正和纹理扭曲，以及创建全景图像。我们将频繁地使用单应性变换。本质上，单应性变换 $\boldsymbol{H}$，按照下面的方程映射二维中的点（齐次坐标意义下）：

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} \quad \text{或} \quad \mathbf{x}' = \boldsymbol{H}\mathbf{x}$$

对于图像平面内（甚至是三维中的点，后面我们会介绍到）的点，齐次坐标是个非常有用的表示方式。点的齐次坐标是依赖于其尺度定义的，所以，$\mathbf{x}=[x,y,w]=[\alpha x,\alpha y,\alpha w]=[x/w,y/w,1]$ 都表示同一个二维点。因此，单应性矩阵 $\boldsymbol{H}$ 也仅依赖尺度定义，所以，单应性矩阵具有 8 个独立的自由度。我们通常使用 $w=1$ 来归一化点，这样，点具有唯一的图像坐标 x 和 y。这个额外的坐标使得我们可以简单地使用一个矩阵来表示变换。

创建 homography.py 文件。下面的函数可以实现对点进行归一化和转换齐次坐标的

功能，将其添加到 homography.py 文件中：

```
def normalize(points):
  """ 在齐次坐标意义下，对点集进行归一化，使最后一行为 1 """
    for row in points:
      row /= points[-1]
    return points

def make_homog(points):
  """ 将点集（dim×n 的数组）转换为齐次坐标表示 """

  return vstack((points,ones((1,points.shape[1]))))
```

进行点和变换的处理时，我们会按照列优先的原则存储这些点。因此，n 个二维点集将会存储为齐次坐标意义下的一个 $3\times n$ 数组。这种格式使得矩阵乘法和点的变换操作更加容易。对于其他的例子，比如对于聚类和分类的特征，我们将使用典型的行数组来存储数据。

在这些投影变换中，有一些特别重要的变换。比如，仿射变换：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & t_x \\ a_3 & a_4 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{或} \quad \boldsymbol{x}' = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix} \boldsymbol{x}$$

保持了 w=1, 不具有投影变换所具有的强大变形能力。仿射变换包含一个可逆矩阵 $\boldsymbol{A}$ 和一个平移向量 $\boldsymbol{t}=[t_x,t_y]$。仿射变换可以用于很多应用，比如图像扭曲。

相似变换：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s\cos(\theta) & -s\sin(\theta) & t_x \\ s\sin(\theta) & s\cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{或} \quad \boldsymbol{x}' = \begin{bmatrix} s\boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix} \boldsymbol{x}$$

是一个包含尺度变化的二维刚体变换。上式中的向量 $\boldsymbol{s}$ 指定了变换的尺度，$\boldsymbol{R}$ 是角度为 θ 的旋转矩阵，$\boldsymbol{t}=[t_x,t_y]$ 在这里也是一个平移向量。如果 $\boldsymbol{s}$=1，那么该变换能够保持距离不变。此时，变换为刚体变换。相似变换可以用于很多应用，比如图像配准。

下面让我们一起探讨如何设计用于估计单应性矩阵的算法，然后看一下使用仿射变换进行图像扭曲，使用相似变换进行图像匹配，以及使用完全投影变换进行创建全景图像的一些例子。

3.1.1 直接线性变换算法

单应性矩阵可以由两幅图像（或者平面）中对应点对计算出来。前面已经提到过，一个完全射影变换具有 8 个自由度。根据对应点约束，每个对应点对可以写出两个方程，分别对应于 x 和 y 坐标。因此，计算单应性矩阵 $\boldsymbol{H}$ 需要 4 个对应点对。

DLT（Direct Linear Transformation，直接线性变换）是给定 4 个或者更多对应点对矩阵，来计算单应性矩阵 $\boldsymbol{H}$ 的算法。将单应性矩阵 $\boldsymbol{H}$ 作用在对应点对上，重新写出该方程，我们可以得到下面的方程：

$$\begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x'_1 & y_1x'_1 & x'_1 \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y'_1 & y_1y'_1 & y'_1 \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x'_2 & y_2x'_2 & x'_2 \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y'_2 & y_2y'_2 & y'_2 \\ & \vdots & & \vdots & & \vdots & & \vdots & \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix} = \mathbf{0}$$

或者 $\boldsymbol{Ah}=\mathbf{0}$，其中 $\boldsymbol{A}$ 是一个具有对应点对二倍数量行数的矩阵。将这些对应点对方程的系数堆叠到一个矩阵中，我们可以使用 SVD（Singular Value Decomposition，奇异值分解）算法找到 $\boldsymbol{H}$ 的最小二乘解。下面是该算法的代码。将下面的函数添加到 homography.py 文件中：

```
def H_from_points(fp,tp):
  """ 使用线性 DLT 方法，计算单应性矩阵 H，使 fp 映射到 tp。点自动进行归一化 """

  if fp.shape != tp.shape:
    raise RuntimeError('number of points do not match')

  # 对点进行归一化（对数值计算很重要）
  # --- 映射起始点 ---
  m = mean(fp[:2], axis=1)
  maxstd = max(std(fp[:2], axis=1)) + 1e-9
  C1 = diag([1/maxstd, 1/maxstd, 1])
  C1[0][2] = -m[0]/maxstd
  C1[1][2] = -m[1]/maxstd
  fp = dot(C1,fp)

  # --- 映射对应点 ---
  m = mean(tp[:2], axis=1)
  maxstd = max(std(tp[:2], axis=1)) + 1e-9
```

```
    C2 = diag([1/maxstd, 1/maxstd, 1])
    C2[0][2] = -m[0]/maxstd
    C2[1][2] = -m[1]/maxstd
    tp = dot(C2,tp)

    # 创建用于线性方法的矩阵，对于每个对应对，在矩阵中会出现两行数值
    nbr_correspondences = fp.shape[1]
    A = zeros((2*nbr_correspondences,9))
    for i in range(nbr_correspondences):
       A[2*i] = [-fp[0][i],-fp[1][i],-1,0,0,0,
             tp[0][i]*fp[0][i],tp[0][i]*fp[1][i],tp[0][i]]
       A[2*i+1] = [0,0,0,-fp[0][i],-fp[1][i],-1,
             tp[1][i]*fp[0][i],tp[1][i]*fp[1][i],tp[1][i]]

    U,S,V = linalg.svd(A)
    H = V[8].reshape((3,3))

    # 反归一化
    H = dot(linalg.inv(C2),dot(H,C1))

    # 归一化，然后返回
    return H / H[2,2]
```

上面函数的第一步操作是检查点对的两个数组中点的数目是否相同。如果不相同，函数将会抛出异常信息。这对于写出稳健的代码来说非常有用。但是，为了使得代码例子更简单、更容易理解，我们在本书中仅在很少的例子中使用异常处理技巧。你可以在 http://docs.python.org/library/exceptions.html 查阅更多关于异常类型的内容，以及在 http://docs.python.org/tutorial/errors.html 上了解如何使用它们。

对这些点进行归一化操作，使其均值为 0，方差为 1。因为算法的稳定性取决于坐标的表示情况和部分数值计算的问题，所以归一化操作非常重要。接下来我们使用对应点对来构造矩阵 $\boldsymbol{A}$。最小二乘解即为矩阵 SVD 分解后所得矩阵 $\boldsymbol{V}$ 的最后一行。该行经过变形后得到矩阵 $\boldsymbol{H}$。然后对这个矩阵进行处理和归一化，返回输出。

3.1.2 仿射变换

由于仿射变换具有 6 个自由度，因此我们需要三个对应点对来估计矩阵 $\boldsymbol{H}$。通过将最后两个元素设置为 0，即 $h_7=h_8=0$，仿射变换可以用上面的 DLT 算法估计得出。

这里我们将使用不同的方法来计算单应性矩阵 $\boldsymbol{H}$，这在文献 [13] 中有详细的描述（第 130 页）。下面的函数使用对应点对来计算仿射变换矩阵，将其添加到 homograph.py 文件中：

```
def Haffine_from_points(fp,tp):
  """ 计算 H，仿射变换，使得 tp 是 fp 经过仿射变换 H 得到的 """

  if fp.shape != tp.shape:
    raise RuntimeError('number of points do not match')

  # 对点进行归一化
  # --- 映射起始点 ---
  m = mean(fp[:2], axis=1)
  maxstd = max(std(fp[:2], axis=1)) + 1e-9
  C1 = diag([1/maxstd, 1/maxstd, 1])
  C1[0][2] = -m[0]/maxstd
  C1[1][2] = -m[1]/maxstd
  fp_cond = dot(C1,fp)

  # --- 映射对应点 ---
  m = mean(tp[:2], axis=1)
  C2 = C1.copy() # 两个点集，必须都进行相同的缩放
  C2[0][2] = -m[0]/maxstd
  C2[1][2] = -m[1]/maxstd
  tp_cond = dot(C2,tp)

  # 因为归一化后点的均值为 0，所以平移量为 0
  A = concatenate((fp_cond[:2],tp_cond[:2]), axis=0)
  U,S,V = linalg.svd(A.T)

  # 如 Hartley 和 Zisserman 著的 Multiple View Geometry in Computer, Scond Edition 所示，
  # 创建矩阵 B 和 C
  tmp = V[:2].T
  B = tmp[:2]
  C = tmp[2:4]

  tmp2 = concatenate((dot(C,linalg.pinv(B)),zeros((2,1))), axis=1)
  H = vstack((tmp2,[0,0,1]))

  # 反归一化
  H = dot(linalg.inv(C2),dot(H,C1))

  return H / H[2,2]
```

同样地，类似于 DLT 算法，这些点需要经过预处理和去处理化操作。在下一节中，让我们一起来看这些仿射变换是如何处理图像的。

3.2 图像扭曲

对图像块应用仿射变换，我们将其称为图像扭曲（或者仿射扭曲）。该操作不仅经

常应用在计算机图形学中，而且经常出现在计算机视觉算法中。扭曲操作可以使用 SciPy 工具包中的 ndimage 包来简单完成。命令：

```
transformed_im = ndimage.affine_transform(im,A,b,size)
```

使用如上所示的一个线性变换 ***A*** 和一个平移向量 ***b*** 来对图像块应用仿射变换。选项参数 size 可以用来指定输出图像的大小。默认输出图像设置为和原始图像同样大小。为了研究该函数是如何工作的，我们可以试着运行下面的命令：

```
from scipy import ndimage

im = array(Image.open('empire.jpg').convert('L'))
H = array([[1.4,0.05,-100],[0.05,1.5,-100],[0,0,1]])
im2 = ndimage.affine_transform(im,H[:2,:2],(H[0,2],H[1,2]))

figure()
gray()
imshow(im2)
show()
```

该命令输出结果图像如图 3-1（右）所示。可以看到，输出图像结果中丢失的像素用零来填充。



图 3-1：用仿射变换扭曲图像。原始图像（左图）以及使用 ndimage.affine_transform() 函数扭曲后的图像（右图）

3.2.1 图像中的图像

仿射扭曲的一个简单例子是，将图像或者图像的一部分放置在另一幅图像中，使得它们能够和指定的区域或者标记物对齐。

将函数 image_in_image() 添加到 warp.py 文件中。该函数的输入参数为两幅图像和一个坐标。该坐标为将第一幅图像放置到第二幅图像中的角点坐标：

```
def image_in_image(im1,im2,tp):
  """ 使用仿射变换将 im1 放置在 im2 上，使 im1 图像的角和 tp 尽可能的靠近
    tp 是齐次表示的，并且是按照从左上角逆时针计算的 """

  # 扭曲的点
  m,n = im1.shape[:2]
  fp = array([[0,m,m,0],[0,0,n,n],[1,1,1,1]])

  # 计算仿射变换，并且将其应用于图像 im1
  H = homography.Haffine_from_points(tp,fp)
  im1_t = ndimage.affine_transform(im1,H[:2,:2],
          (H[0,2],H[1,2]),im2.shape[:2])
  alpha = (im1_t > 0)

  return (1-alpha)*im2 + alpha*im1_t
```

正如你所看到的，该函数没有很多繁杂的操作。将扭曲的图像和第二幅图像融合，我们就创建了 alpha 图像。该图像定义了每个像素从各个图像中获取的像素值成分多少。这里我们基于以下事实，扭曲的图像是在扭曲区域边界之外以 0 来填充的图像，来创建一个二值的 alpha 图像。严格意义上说，我们需要在第一幅图像中的潜在 0 像素上加上一个小的数值，或者合理地处理这些 0 像素（参见本章结尾的练习部分）。注意，这里我们使用的图像坐标是齐次坐标意义下的。

试着使用该函数将公告牌中的一幅图像插入另一幅图像。下面几行代码会将图 3-2 中最左端的图像插入到第二幅图像中。这些坐标值是通过查看绘制的图像（在 PyLab 图像中，鼠标的坐标显示在图像底部附近）手工确定的。当然，也可以用 PyLab 类库中的 ginput() 函数获得。



图 3-2：使用仿射变换将一幅图像放置到另一幅图像中

```
import warp

# 仿射扭曲 im1 到 im2 的例子
im1 = array(Image.open('beatles.jpg').convert('L'))
im2 = array(Image.open('billboard_for_rent.jpg').convert('L'))

# 选定一些目标点
tp = array([[264,538,540,264],[40,36,605,605],[1,1,1,1]])

im3 = warp.image_in_image(im1,im2,tp)

figure()
gray()
imshow(im3)
axis('equal')
axis('off')
show()
```

上面的代码将图像放置在公告牌的上半部分。需要注意，标记物的坐标 tp 是用齐次坐标意义下的坐标表示的。将这些坐标换成：

```
tp = array([[675,826,826,677],[55,52,281,277],[1,1,1,1]])
```

会将图像放置在公告牌的左下“for rent”部分。

函数 Haffine_from_points() 会返回给定对应点对的最优仿射变换。在上面的例子中，对应点对为图像和公告牌的角点。如果透视效应比较弱，那么这种方法会返回很好的结果。图 3-3 的上面一行显示出，在具有很强透视效应的情况下，在公告牌图像上使用射影变换输出图像的情况。在这种情况下，我们不可能使用同一个仿射变换将全部 4 个角点变换到它们的目标位置（尽管我们可以使用完全投影变换来完成该任务）。所以，当你打算使用仿射变换时，有一个很有用的技巧。



图 3-3：比较完全图像的仿射扭曲和使用两个三角形的仿射弯曲效果。图像放置在公告牌上，并且带有一些透视效应。对于整幅图像使用仿射扭曲效果不好。为了看清楚，我们将右边的两个角点放大（上）。使用包含两个三角形的仿射变换可以很好地将图像完全放置到公告牌上（下）

对于三个点，仿射变换可以将一幅图像进行扭曲，使这三对对应点对可以完美地匹配上。这是因为，仿射变换具有 6 个自由度，三个对应点对可以给出 6 个约束条件（对于这三个对应点对，x 和 y 坐标必须都要匹配）。所以，如果你真的打算使用仿射变换将图像放置到公告牌上，可以将图像分成两个三角形，然后对它们分别进行扭曲图像操作。下面是具体实现的代码：

```
# 选定 im1 角上的一些点
m,n = im1.shape[:2]
fp = array([[0,m,m,0],[0,0,n,n],[1,1,1,1]])

# 第一个三角形
tp2 = tp[:,:3]
fp2 = fp[:,:3]

# 计算 H
H = homography.Haffine_from_points(tp2,fp2)
im1_t = ndimage.affine_transform(im1,H[:2,:2],
          (H[0,2],H[1,2]),im2.shape[:2])

# 三角形的 alpha
alpha = warp.alpha_for_triangle(tp2,im2.shape[0],im2.shape[1])
```

```
im3 = (1-alpha)*im2 + alpha*im1_t

# 第二个三角形
tp2 = tp[:,[0,2,3]]
fp2 = fp[:,[0,2,3]]

# 计算 H
H = homography.Haffine_from_points(tp2,fp2)
im1_t = ndimage.affine_transform(im1,H[:2,:2],
          (H[0,2],H[1,2]),im2.shape[:2])

# 三角形的 alpha 图像
alpha = warp.alpha_for_triangle(tp2,im2.shape[0],im2.shape[1])
im4 = (1-alpha)*im3 + alpha*im1_t

figure()
gray()
imshow(im4)
axis('equal')
axis('off')
show()
```

这里我们简单地为每个三角形创建了 alpha 图像，然后将所有的图像合并起来。该三角形的 alpha 图像可以简单地通过检查像素的坐标是否能够写成三角形顶点坐标的凸组合来计算得出[1]。如果坐标可以表示成这种形式，那么该像素就位于三角形的内部。上面的例子使用了下面的函数 `alpha_for_triangle()`，将其添加到 warp.py 文件中。

```
def alpha_for_triangle(points,m,n):
  """ 对于带有由 points 定义角点的三角形，创建大小为 (m，n) 的 alpha 图
    （在归一化的齐次坐标意义下）"""

  alpha = zeros((m,n))
    for i in range(min(points[0]),max(points[0])):
      for j in range(min(points[1]),max(points[1])):
        x = linalg.solve(points,[i,j,1])
        if min(x) > 0: # 所有系数都大于零
            alpha[i,j] = 1
  return alpha
```

你的显卡可以极其快速地操作上面的代码。Python 语言的处理速度比你的显卡（或者 C/C++ 实现）慢很多，但是对于我们来说已经够用了。正如在图 3-3 下半部分所看到的，角点可以很好地匹配。

注 1：凸组合是形式为 $\sum_j \alpha_j x_i$ 的线性组合（在三角形的例子中），其中所有的系数 α_j 非负，并且和为 1。

3.2.2 分段仿射扭曲

正如上面的例子所示，三角形图像块的仿射扭曲可以完成角点的精确匹配。让我们看一下对应点对集合之间最常用的扭曲方式：*分段仿射扭曲*。给定任意图像的标记点，通过将这些点进行三角剖分，然后使用仿射扭曲来扭曲每个三角形，我们可以将图像和另一幅图像的对应标记点扭曲对应。对于任何图形和图像处理库来说，这些都是最基本的操作。下面我们来演示一下如何使用 Matplotlib 和 SciPy 来完成该操作。

为了三角化这些点，我们经常使用*狄洛克三角剖分方法*。在 Matplotlib（但是不在 PyLab 库中）中有狄洛克三角剖分，我们可以用下面的方式使用它：

```
import matplotlib.delaunay as md

x,y = array(random.standard_normal((2,100)))
centers,edges,tri,neighbors = md.delaunay(x,y)

figure()
for t in tri:
    t_ext = [t[0], t[1], t[2], t[0]] # 将第一个点加入到最后
    plot(x[t_ext],y[t_ext],'r')

plot(x,y,'*')
axis('off')
show()
```

图 3-4 显示了一些实例点和三角剖分的结果。狄洛克三角剖分选择一些三角形，使三角剖分中所有三角形的最小角度最大[1]。函数 delaunay() 有 4 个输出，其中我们仅需要三角形列表信息（第三个输出）。在 warp.py 文件中创建用于三角剖分的函数：

```
import matplotlib.delaunay as md

def triangulate_points(x,y):
  """ 二维点的 Delaunay 三角剖分 """

  centers,edges,tri,neighbors = md.delaunay(x,y)
  return tri
```

函数输出的是一个数组，该数组的每一行包含对应数组 x 和 y 中每个三角形三个点的切片。

注 1：三角剖分中的边实际上是泰森图的对偶图。参见 http://en.wikipedia.org/wiki/Delaunay_triangulation。

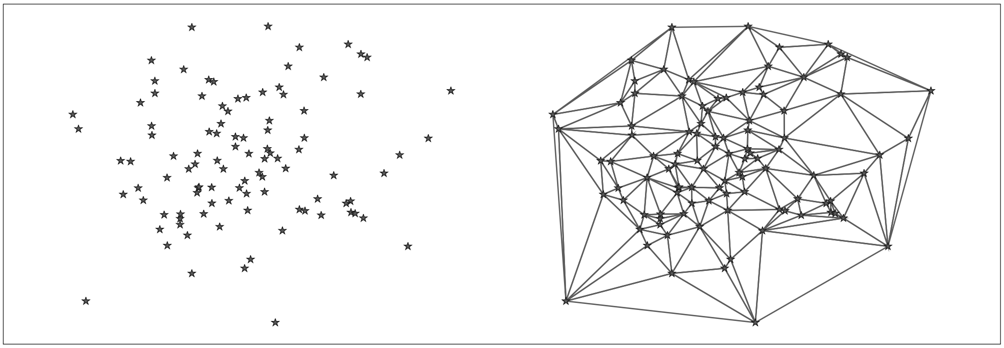


图 3-4：随机二维点集的狄洛克三角剖分示例

现在让我们将该算法应用于一个例子，在该例子中，在 5×6 的网格上使用 30 个控制点，将一幅图像扭曲到另一幅图像中的非平坦区域。图 3-5b 所示的是将一幅图像扭曲到“turning torso”的表面。目标点是使用 ginput() 函数手工选取出来的，将结果保存在 turningtorso_points.txt 文件中。

首先，我们需要写出一个用于分段仿射图像扭曲的通用扭曲函数。下面的代码可以实现该功能。在该代码中，我们也展示了如何扭曲一幅彩色图像（你仅需要对每个颜色通道进行扭曲）。

```
def pw_affine(fromim,toim,fp,tp,tri):
  """ 从一幅图像中扭曲矩形图像块
    fromim= 将要扭曲的图像
    toim= 目标图像
    fp= 齐次坐标表示下，扭曲前的点
    tp= 齐次坐标表示下，扭曲后的点
    tri= 三角剖分 """

  im = toim.copy()

  # 检查图像是灰度图像还是彩色图象
  is_color = len(fromim.shape) == 3

  # 创建扭曲后的图像（如果需要对彩色图像的每个颜色通道进行迭代操作，那么有必要这样做）
  im_t = zeros(im.shape, 'uint8')

  for t in tri:
    # 计算仿射变换
    H = homography.Haffine_from_points(tp[:,t],fp[:,t])

    if is_color:
      for col in range(fromim.shape[2]):
```

```
        im_t[:,:,col] = ndimage.affine_transform(
          fromim[:,:,col],H[:2,:2],(H[0,2],H[1,2]),im.shape[:2])
    else:
      im_t = ndimage.affine_transform(
        fromim,H[:2,:2],(H[0,2],H[1,2]),im.shape[:2])

    # 三角形的 alpha
    alpha = alpha_for_triangle(tp[:,t],im.shape[0],im.shape[1])

    # 将三角形加入到图像中
    im[alpha>0] = im_t[alpha>0]

  return im
```

在该代码中，我们首先检查该图像是灰度图像还是彩色图像。如果图像为彩色图像，则对每个颜色通道进行扭曲处理。因为对于每个三角形来说，仿射变换是唯一确定的，所以我们这里使用 Haffine_from_points() 函数来处理。将上面的函数添加到 warp.py 文件中。

为了将该函数应用到当前例子中，接下来的简短脚本将这些操作统一起来：

```
import homography
import warp

# 打开图像，并将其扭曲
fromim = array(Image.open('sunset_tree.jpg'))
x,y = meshgrid(range(5),range(6))
x = (fromim.shape[1]/4) * x.flatten()
y = (fromim.shape[0]/5) * y.flatten()

# 三角剖分
tri = warp.triangulate_points(x,y)

# 打开图像和目标点
im = array(Image.open('turningtorso1.jpg'))
tp = loadtxt('turningtorso1_points.txt') # destination points

# 将点转换成齐次坐标
fp = vstack((y,x,ones((1,len(x)))))
tp = vstack((tp[:,1],tp[:,0],ones((1,len(tp)))))

# 扭曲三角形
im = warp.pw_affine(fromim,im,fp,tp,tri)
```

```
# 绘制图像
figure()
imshow(im)
warp.plot_mesh(tp[1],tp[0],tri)
axis('off')
show()
```

输出结果如图 3-5c 所示。我们通过下面的辅助函数（将其添加到 warp.py 文件中）来绘制出图像中的这些三角形：

```
def plot_mesh(x,y,tri):
  """ 绘制三角形 """

  for t in tri:
    t_ext = [t[0], t[1], t[2], t[0]] # 将第一个点加入到最后
    plot(x[t_ext],y[t_ext],'r')
```

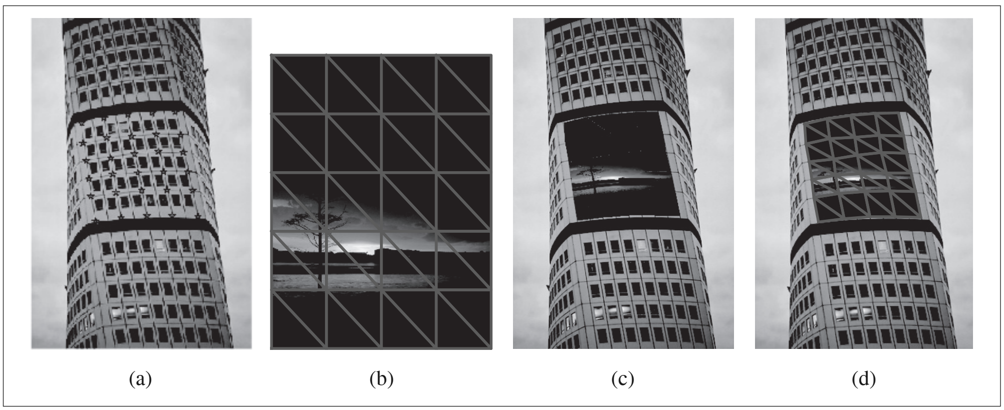


图 3-5：使用狄洛克三角剖分标记点进行分段仿射扭曲：（a）为带有标记物的目标图像；（b）为带有三角剖分的图像；（c）为扭曲后的图像；（d）为带有三角剖分的扭曲图像

这个例子应该能够帮助你在应用中做图像的分段仿射扭曲。我们可以对该例子中的函数进行改进。我们将其中一部分留作练习，剩下的留给你自己解决。

3.2.3 图像配准

图像配准是对图像进行变换，使变换后的图像能够在常见的坐标系中对齐。配准可以是严格配准，也可以是非严格配准。为了能够进行图像对比和更精细的图像分析，图像配准是一步非常重要的操作。

让我们一起看一个对多个人脸图像进行严格配准的例子。该配准使得我们计算的平

均人脸和人脸表观的变化具有意义。因为，图像中的人脸并不都有相同的大小、位置和方向，所以，在这种类型的配准中，我们实际上是寻找一个相似变换（带有尺度变化的刚体变换），在对应点对之间建立映射。

在 jkface.zip 文件中有 366 幅单人图像（2008 年，每天一幅）。[1] 这些图像都对眼睛和嘴的坐标进行了标记，结果保存在 jkface.xml 文件中。使用这些点，我们可以计算出一个相似变换，然后将可以使用该变换（包含尺度变换）的这些图像扭曲到一个归一化的坐标系中。为了读取 XML 格式的文件，我们将会使用 Python 中内置 `xml.dom` 模块中的 `minidom` 类库。

该 XML 文件看起来类似于下面的格式：

```
<?xml version="1.0" encoding="utf-8"?>
<faces>
        <face file="jk-002.jpg" xf="46" xm="56" xs="67" yf="38" ym="65" ys="39"/>
        <face file="jk-006.jpg" xf="38" xm="48" xs="59" yf="38" ym="65" ys="38"/>
        <face file="jk-004.jpg" xf="40" xm="50" xs="61" yf="38" ym="66" ys="39"/>
        <face file="jk-010.jpg" xf="33" xm="44" xs="55" yf="38" ym="65" ys="38"/>
⋮
</faces>
```

为了从该文件中读取这些坐标，我们需要将使用 `minidom` 的函数添加到新文件 imregistration.py 中：

```
from xml.dom import minidom

def read_points_from_xml(xmlFileName):
  """ 读取用于人脸对齐的控制点 """

  xmldoc = minidom.parse(xmlFileName)
  facelist = xmldoc.getElementsByTagName('face')
  faces = {}
  for xmlFace in facelist:
    fileName = xmlFace.attributes['file'].value
    xf = int(xmlFace.attributes['xf'].value)
    yf = int(xmlFace.attributes['yf'].value)
    xs = int(xmlFace.attributes['xs'].value)
    ys = int(xmlFace.attributes['ys'].value)
    xm = int(xmlFace.attributes['xm'].value)
    ym = int(xmlFace.attributes['ym'].value)
    faces[fileName] = array([xf, yf, xs, ys, xm, ym])
  return faces
```

注 1：这些图像是由 J. K. Keller（经过许可）提供的，详情参见 http://jk-keller.com/daily-photo/。

这些标记点会在 Python 中以字典的形式返回，字典的键值为图像的文件名。格式为：图像中左眼（人脸右侧）的坐标为 xf 和 yf，右眼的坐标为 xs 和 ys，嘴的坐标为 xm 和 ym。为了计算相似变换中的参数，我们可以使用最小二乘解来解决。对于每个点 $\mathbf{x}_i=[x_i, y_i]$（在这个例子中，每幅图像有三个点），这些点应该被映射到目标位置 $[\hat{x}_i, \hat{y}_i]$，如下所示：

$$\begin{bmatrix} \hat{x}_i \\ \hat{y}_i \end{bmatrix} = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

将这三个点都表示成该形式，我们可以重新将其写成方程组的形式。该方程组中含有 a、b、t_x、t_y 未知量，如下所示：

$$\begin{bmatrix} \hat{x}_1 \\ \hat{y}_1 \\ \hat{x}_2 \\ \hat{y}_2 \\ \hat{x}_3 \\ \hat{y}_3 \end{bmatrix} = \begin{bmatrix} x_1 & -y_1 & 1 & 0 \\ y_1 & x_1 & 0 & 1 \\ x_2 & -y_2 & 1 & 0 \\ y_2 & x_2 & 0 & 1 \\ x_3 & -y_3 & 1 & 0 \\ y_3 & x_3 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ t_x \\ t_y \end{bmatrix}$$

下面我们使用相似矩阵的参数化表示方式：

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} = s \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} = s\boldsymbol{R}$$

其中尺度 $s = \sqrt{a^2 + b^2}$，旋转矩阵为 $\boldsymbol{R}$。

如果存在更多的对应点对，其计算公式相同，只需在矩阵中额外添加几行。你可以使用 `linalg.lstsq()` 函数来计算该问题的最小二乘解。 使用最小二乘解的思想是一个标准技巧，我们还会在本书中多次使用。实际上，这和之前在 DLT 算法中使用的方式相同。

函数的具体代码如下（将其添加到 imregistration.py 文件中）：

```
from scipy import linalg

def compute_rigid_transform(refpoints,points):
  """ 计算用于将点对齐到参考点的旋转、尺度和平移量 """

  A = array([ [points[0], -points[1], 1, 0],
              [points[1],  points[0], 0, 1],
              [points[2], -points[3], 1, 0],
              [points[3],  points[2], 0, 1],
```

```
                [points[4], -points[5], 1, 0],
                [points[5],  points[4], 0, 1]])

    y = array([ refpoints[0],
                refpoints[1],
                refpoints[2],
                refpoints[3],
                refpoints[4],
                refpoints[5]])

    # 计算最小化 ||Ax-y|| 的最小二乘解
    a,b,tx,ty = linalg.lstsq(A,y)[0]
    R = array([[a, -b], [b, a]]) # 包含尺度的旋转矩阵

    return R,tx,ty
```

该函数返回一个具有尺度的旋转矩阵，以及在 x 和 y 方向上的平移量。为了扭曲图像，并保存对齐后的新图像，我们可以对每个颜色通道（这些图像都是彩色图像）应用 ndimage.affine_transform() 函数操作。作为参考坐标系，你可以使用任何三个点的坐标。这里我们为了简单起见，直接使用第一幅图像中的标记位置：

```
from scipy import ndimage
from scipy.misc import imsave
import os

def rigid_alignment(faces,path,plotflag=False):
  """ 严格对齐图像，并将其保存为新的图像
    path 决定对齐后图像保存的位置
    设置 plotflag=True，以绘制图像 """

  # 将第一幅图像中的点作为参考点
  refpoints = faces.values()[0]

  # 使用仿射变换扭曲每幅图像
  for face in faces:
    points = faces[face]
  R,tx,ty = compute_rigid_transform(refpoints, points)
  T = array([[R[1][1], R[1][0]], [R[0][1], R[0][0]]])

  im = array(Image.open(os.path.join(path,face)))
  im2 = zeros(im.shape, 'uint8')

  # 对每个颜色通道进行扭曲
  for i in range(len(im.shape)):
    im2[:,:,i] = ndimage.affine_transform(im[:,:,i],linalg.inv(T),offset=[-ty,-tx])
```

```
  if plotflag:
    imshow(im2)
    show()

  # 裁剪边界，并保存对齐后的图像
  h,w = im2.shape[:2]
  border = (w+h)/20

  # 裁剪边界
  imsave(os.path.join(path, 'aligned/'+face),im2[border:h-border,border:w-border,:])
```

这里我们使用 imsave() 函数来将对齐后的图像保存到 aligned 子文件夹中。

接下来的简短脚本会读取 XML 文件，其中文件名为键，点的坐标为键值。然后配准所有的图像，将它们与第一幅图像对齐：

```
import imregistration

# 载入控制点的位置
xmlFileName = 'jkfaces2008_small/jkfaces.xml'
points = imregistration.read_points_from_xml(xmlFileName)

# 注册
imregistration.rigid_alignment(points,'jkfaces2008_small/')
```

运行这些代码，你能够在子目录中得到这些对齐后的人脸图像。图 3-6 所示为配准前后的 6 幅样本图像。由于配准后图像的边界可能会出现不想要的黑色填充像素，所以我们对配准后的图像进行轻微的修剪，来删除这些黑色填充像素。

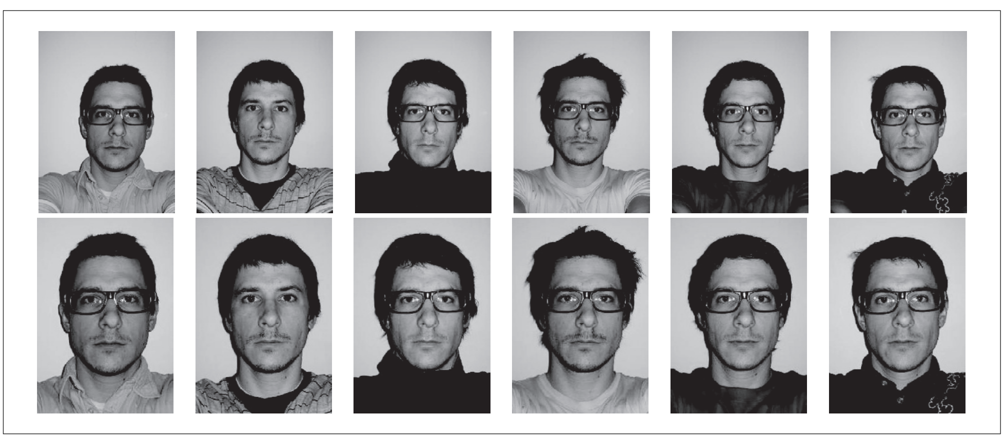


图 3-6：配准之前（上）和配准之后（下）的样本图像

现在让我们看配准操作如何影响平均图像。图 3-7 为未对齐人脸图像的平均图像，旁边是对齐后图像的平均图像。（注意，由于对齐后图像的边界有裁剪，所以两幅图像的大小有差异）尽管在原始图像中，人脸的尺寸、方向和位置变化都很小，但是配准操作对平均图像的计算结果影响很大。



图 3-7：平均图像的比较：没有对齐操作（左）；经过三点刚体对齐操作（右）

自然地，使用未准确配准的图像同样对主成分的计算结果有着很大的影响。图 3-8 表示，从未经过配准和经过配准的数据集中选取前 150 幅图像，PCA 的计算结果。正如平均图像一样，未配准的 PCA 模式是模糊的。在计算主成分时，我们使用以平均人脸位置为中心的椭圆掩膜。在堆叠这些图像之前，将这些图像和掩膜相乘，我们能够避免将背景变化带入到 PCA 模式中。将 1.3 节 PCA 例子中创建矩阵的一行替换为：

```
immatrix = array([mask*array(Image.open(imlist[i]).convert('L')).flatten()
          for i in range(150)],'f')
```

其中 mask 是一副同样大小的二值图像，已经经过压平处理。



图 3-8：比较未配准和已配准图像的 PCA 模式：未经过配准的平均图像和前 9 个主成分（上）；经过配准后的平均图像和前 9 个主成分（下）

3.3 创建全景图

在同一位置（即图像的照相机位置相同）拍摄的两幅或者多幅图像是单应性相关的（如图 3-9 所示）。我们经常使用该约束将很多图像缝补起来，拼成一个大的图像来创建全景图像。在本节中，我们将探讨如何创建全景图像。



图 3-9：瑞典隆德主要大学建筑的 5 幅图像。这些图像都是从同一个视点拍摄的

3.3.1 RANSAC

RANSAC 是“RANdom SAmple Consensus”（随机一致性采样）的缩写。该方法是用来找到正确模型来拟合带有噪声数据的迭代方法。给定一个模型，例如点集之间的单应性矩阵，RANSAC 基本的思想是，数据中包含正确的点和噪声点，合理的模型应该能够在描述正确数据点的同时摒弃噪声点。

RANSAC 的标准例子：用一条直线拟合带有噪声数据的点集。简单的最小二乘在该例子中可能会失效，但是 RANSAC 能够挑选出正确的点，然后获取能够正确拟合的直线。下面来看使用 RANSAC 的例子。你可以从 http://www.scipy.org/Cookbook/RANSAC 下载 ransac.py，里面包含了特定的例子作为测试用例。图 3-10 为运行 `ransac.text()` 的例子。可以看到，该算法只选择了和直线模型一致的数据点，成功地找到了正确的解。

RANSAC 是个非常有用的算法，我们将在下一节估计单应性矩阵和其他一些例子中使用它。关于 RANSAC 更多的信息，参见 Fischler 和 Bolles 的原始论文 [11]、维基百科 http://en.wikipedia.org/wiki/RANSAC 或者技术报告 [40]。

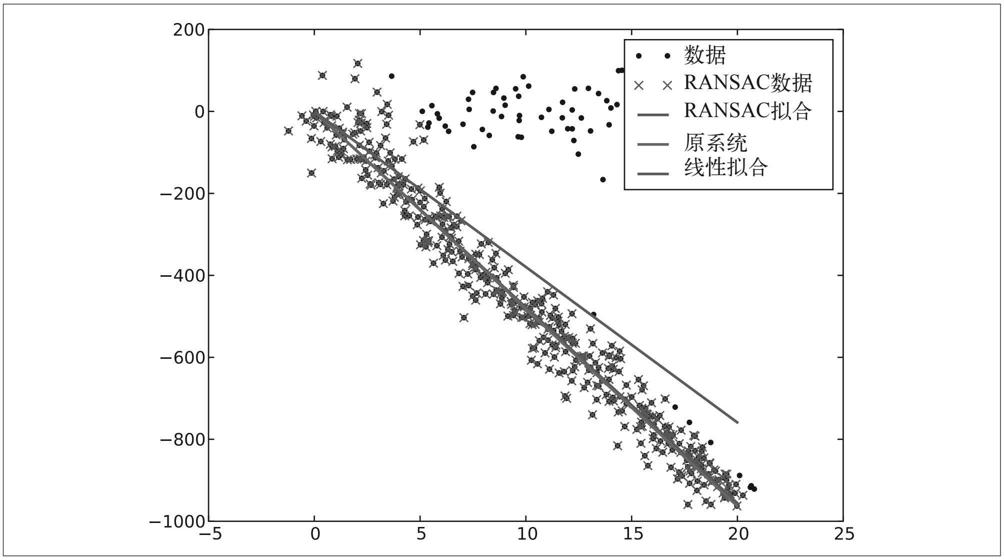


图 3-10：使用 RANSAC 算法用一条直线来拟合包含噪声数据点集

3.3.2 稳健的单应性矩阵估计

我们在任何模型中都可以使用 RANSAC 模块。在使用 RANSAC 模块时，我们只需要在相应 Python 类中实现 `fit()` 和 `get_error()` 方法，剩下就是正确地使用 ransac.py。我们这里使用可能的对应点集来自动找到用于全景图像的单应性矩阵。图 3-11 所示为使用 SIFT 特征自动找到匹配对应。这可以通过运行下面的命令来实现：

```
import sift

featname = ['Univ'+str(i+1)+'.sift' for i in range(5)]
imname = ['Univ'+str(i+1)+'.jpg' for i in range(5)]
l = {}
d = {}
for i in range(5):
  sift.process_image(imname[i],featname[i])
  l[i],d[i] = sift.read_features_from_file(featname[i])

matches = {}
for i in range(4):
  matches[i] = sift.match(d[i+1],d[i])
```

显然，并不是所有图像中的对应点对都是正确的。实际上，SIFT 是具有很强稳健性的描述子，能够比其他描述子，例如图像块相关的 Harris 角点，产生更少的错误的匹配。但是该方法仍然远非完美。



图 3-11：在连续图像对间使用 SIFT 特征寻找匹配对应点对

我们使用 RANSAC 算法来求解单应性矩阵，首先需要将下面模型类添加到 homography.py 文件中：

```
class RansacModel(object):
  """ 用于测试单应性矩阵的类，其中单应性矩阵是由网站 http://www.scipy.org/Cookbook/RANSAC 上的 ransac.py 计算出来的 """

  def __init__(self,debug=False):
    self.debug = debug

  def fit(self, data):
  """ 计算选取的 4 个对应的单应性矩阵 """

  # 将其转置，来调用 H_from_points() 计算单应性矩阵
  data = data.T

  # 映射的起始点
  fp = data[:3,:4]
  # 映射的目标点
  tp = data[3:,:4]
```

```
        # 计算单应性矩阵，然后返回
        return H_from_points(fp,tp)

    def get_error( self, data, H):
        """ 对所有的对应计算单应性矩阵，然后对每个变换后的点，返回相应的误差 """

        data = data.T

        # 映射的起始点
        fp = data[:3]
        # 映射的目标点
        tp = data[3:]

        # 变换 fp
        fp_transformed = dot(H,fp)

        # 归一化齐次坐标
        for i in range(3):
          fp_transformed[i] /= fp_transformed[2]

        # 返回每个点的误差
        return sqrt( sum((tp-fp_transformed)**2,axis=0) )
```

可以看到，这个类包含 fit() 方法。该方法仅仅接受由 ransac.py 选择的 4 个对应点对（data 中的前 4 个点对），然后拟合一个单应性矩阵。记住， 4 个点对是计算单应性矩阵所需的最少数目。由于 get_error() 方法对每个对应点对使用该单应性矩阵，然后返回相应的平方距离之和，因此 RANSAC 算法能够判定哪些点对是正确的，哪些是错误的。在实际中，我们需要在距离上使用一个阈值来决定哪些单应性矩阵是合理的。为了方便使用，将下面的函数添加到 homography.py 文件中：

```
    def H_from_ransac(fp,tp,model,maxiter=1000,match_theshold=10):
      """ 使用 RANSAC 稳健性估计点对应间的单应性矩阵 H（ransac.py 为从
        http://www.scipy.org/Cookbook/RANSAC 下载的版本）

        # 输入：齐次坐标表示的点 fp，tp（3×n 的数组）"""

      import ransac

      # 对应点组
      data = vstack((fp,tp))

      # 计算 H，并返回
      H,ransac_data = ransac.ransac(data.T,model,4,maxiter,match_theshold,10,
                                   return_all=True)
      return H,ransac_data['inliers']
```

该函数同样允许提供阈值和最小期望的点对数目。最重要的参数是最大迭代次数：程序退出太早可能得到一个坏解；迭代次数太多会占用太多时间。函数的返回结果为单应性矩阵和对应该单应性矩阵的正确点对。

类似于下面的操作，你可以将 RANSAC 算法应用于对应点对上：

```
# 将匹配转换成齐次坐标点的函数
def convert_points(j):
  ndx = matches[j].nonzero()[0]
  fp = homography.make_homog(l[j+1][ndx,:2].T)
  ndx2 = [int(matches[j][i]) for i in ndx]
  tp = homography.make_homog(l[j][ndx2,:2].T)
  return fp,tp

# 估计单应性矩阵
model = homography.RansacModel()

fp,tp = convert_points(1)
H_12 = homography.H_from_ransac(fp,tp,model)[0] # im1 到 im2 的单应性矩阵

fp,tp = convert_points(0)
H_01 = homography.H_from_ransac(fp,tp,model)[0] # im0 到 im1 的单应性矩阵

tp,fp = convert_points(2) # 注意：点是反序的
H_32 = homography.H_from_ransac(fp,tp,model)[0] # im3 到 im2 的单应性矩阵

tp,fp = convert_points(3) # 注意：点是反序的
H_43 = homography.H_from_ransac(fp,tp,model)[0] # im4 到 im3 的单应性矩阵
```

在该例子中，图像 2 是中心图像，也是我们希望将其他图像变成的图像。图像 0 和图像 1 应该从右边扭曲，图像 3 和图像 4 从左边扭曲。在每个图像对中，由于匹配是从最右边的图像计算出来的，所以我们将对应的顺序进行了颠倒，使其从左边图像开始扭曲。因为我们不关心该扭曲例子中的正确点对，所以仅需要该函数的第一个输出（单应性矩阵）。

3.3.3 拼接图像

估计出图像间的单应性矩阵（使用 RANSAC 算法），现在我们需要将所有的图像扭曲到一个公共的图像平面上。通常，这里的公共平面为中心图像平面（否则，需要进行大量变形）。一种方法是创建一个很大的图像，比如图像中全部填充 0，使其和中心图像平行，然后将所有的图像扭曲到上面。由于我们所有的图像是由照相机水平旋转拍摄的，因此我们可以使用一个较简单的步骤：将中心图像左边或者右边的区域

填充 0，以便为扭曲的图像腾出空间。将下面的代码添加到 warp.py 文件中：

```
def panorama(H,fromim,toim,padding=2400,delta=2400):
  """ 使用单应性矩阵 H（使用 RANSAC 健壮性估计得出），协调两幅图像，创建水平全景图像。结果
    为一幅和 toim 具有相同高度的图像。padding 指定填充像素的数目，delta 指定额外的平移量 """

  # 检查图像是灰度图像，还是彩色图像
  is_color = len(fromim.shape) == 3

  # 用于 geometric_transform() 的单应性变换
  def transf(p):
    p2 = dot(H,[p[0],p[1],1])
    return (p2[0]/p2[2],p2[1]/p2[2])

  if H[1,2]<0: # fromim 在右边
    print 'warp - right'
    # 变换 fromim
    if is_color:
      # 在目标图像的右边填充 0
      toim_t = hstack((toim,zeros((toim.shape[0],padding,3))))
      fromim_t = zeros((toim.shape[0],toim.shape[1]+padding,toim.shape[2]))
      for col in range(3):
        fromim_t[:,:,col] = ndimage.geometric_transform(fromim[:,:,col],
                    transf,(toim.shape[0],toim.shape[1]+padding))
  else:
    # 在目标图像的右边填充 0
    toim_t = hstack((toim,zeros((toim.shape[0],padding))))
    fromim_t = ndimage.geometric_transform(fromim,transf,
                  (toim.shape[0],toim.shape[1]+padding))
  else:
    print 'warp - left'
    # 为了补偿填充效果，在左边加入平移量
    H_delta = array([[1,0,0],[0,1,-delta],[0,0,1]])
    H = dot(H,H_delta)
    # fromim 变换
    if is_color:
    # 在目标图像的左边填充 0
    toim_t = hstack((zeros((toim.shape[0],padding,3)),toim))
    fromim_t = zeros((toim.shape[0],toim.shape[1]+padding,toim.shape[2]))
    for col in range(3):
      fromim_t[:,:,col] = ndimage.geometric_transform(fromim[:,:,col],
                  transf,(toim.shape[0],toim.shape[1]+padding))
    else:
    # 在目标图像的左边填充 0
    toim_t = hstack((zeros((toim.shape[0],padding)),toim))
    fromim_t = ndimage.geometric_transform(fromim,
```

```
                transf,(toim.shape[0],toim.shape[1]+padding))

    # 协调后返回（将 fromim 放置在 toim 上）
    if is_color:
      # 所有非黑色像素
      alpha = ((fromim_t[:,:,0] * fromim_t[:,:,1] * fromim_t[:,:,2] ) > 0)
      for col in range(3):
        toim_t[:,:,col] = fromim_t[:,:,col]*alpha + toim_t[:,:,col]*(1-alpha)
    else:
      alpha = (fromim_t > 0)
      toim_t = fromim_t*alpha + toim_t*(1-alpha)

    return toim_t
```

对于通用的 geometric_transform() 函数，我们需要指定能够描述像素到像素间映射的函数。在这个例子中，transf() 函数就是该指定的函数。该函数通过将像素和 $\boldsymbol{H}$ 相乘，然后对齐次坐标进行归一化来实现像素间的映射。通过查看 $\boldsymbol{H}$ 中的平移量，我们可以决定应该将该图像填补到左边还是右边。当该图像填补到左边时，由于目标图像中点的坐标也变化了，所以在“左边”情况中，需要在单应性矩阵中加入平移。简单起见，我们同样使用 0 像素的技巧来寻找 alpha 图。

现在在图像中使用该操作，函数如下所示：

```
# 扭曲图像
delta = 2000 # 用于填充和平移

im1 = array(Image.open(imname[1]))
im2 = array(Image.open(imname[2]))
im_12 = warp.panorama(H_12,im1,im2,delta,delta)

im1 = array(Image.open(imname[0]))
im_02 = warp.panorama(dot(H_12,H_01),im1,im_12,delta,delta)

im1 = array(Image.open(imname[3]))
im_32 = warp.panorama(H_32,im1,im_02,delta,delta)

im1 = array(Image.open(imname[j+1]))
im_42 = warp.panorama(dot(H_32,H_43),im1,im_32,delta,2*delta)
```

注意，在最后一行中，im_32 图像已经发生了一次平移。创建的全景图结果如图 3-12 所示。正如你所看到的，图像曝光不同，在单个图像的边界上存在边缘效应。商业的创建全景图像软件里有额外的操作来对强度进行归一化，并对平移进行平滑场景转换，以使得结果看上去更好。



图 3-12：使用 SIFT 对应点对自动创建水平全景图像：全部的全景图像（上）；对中心部分图像进行裁剪后的图像（下）

练习

(1) 写出一个函数，其输入参数为正方形（或者长方形）物体（例如，一本书、一张海报，或者二维条形码）图像的坐标。然后，计算将该长方形映射归一化坐标系中正视图全图的变换。你可以使用 ginput()，或者最强的 Harris 角点来发现长方形物体的稳健性角点。

(2) 写出一个函数，对于如图 3-1 所示的扭曲能够正确地找到 alpha 图像。

(3) 在你自己的数据集中找出包含三个公共的标记物（像人脸例子一样，或者使用著名的景物，比如埃菲尔铁塔）的那个。创建对齐后的图像，其中这些标记物在同一个位置上。计算平均和中值图像，然后可视化。

(4) 进行亮度归一化操作，找出在全景图像例子中更好地拼接图像的方法。该方法能够去除图 3-12 中的边缘效应。

(5) 与将图像扭曲到中心图像上不同，全景图像可以通过将图像扭曲到圆柱体上来创建。试着在图 3-12 的例子中使用该方式创建全景图像。

(6) 使用 RANSAC 算法来找到一些主要的正确单应性矩阵集合。一个简单的方式是，首先运行一次 RANSAC 算法，找到具有最大一致子集的单应性矩阵，然后将与该单应性矩阵一致的对应点对从匹配集合中删除，再运行 RANSAC 算法找到下一个最大的集合，以此类推。

(7) 修改单应性矩阵的 RANSAC 估计算法，来使用三个对应点对计算仿射变换。使用该算法来判断图像对之间是否包含平面场景，例如使用正确点的个数。对于仿射变化，平面场景中正确点的个数会很多。

(8) 通过匹配局部特征，以及使用最小二乘刚体配准，用多个图像（例如，从 Flickr 下载）创建一个全景图（http://en.wikipedia.org/wiki/Panography）。

第 4 章

照相机模型与增强现实

本章中，我们将尝试对照相机进行建模，并有效地使用这些模型。在之前的章节里，我们已经讲述了图像到图像之间的映射和变换。为了处理三维图像和平面图像之间的映射，我们需要在映射中加入部分照相机产生图像过程的投影特性。本章中我们将会讲述如何确定照相机的参数，以及在具体应用中，如增强现实，如何使用图像间的投影变换。下一章中，我们将使用照相机模型处理其他一些应用，比如多视图及其映射。

4.1 针孔照相机模型

针孔照相机模型（有时称为射影照相机模型）是计算机视觉中广泛使用的照相机模型。对于大多数应用来说，针孔照相机模型简单，并且具有足够的精确度。这个名字来源于一种类似暗箱机的照相机。该照相机从一个小孔采集射到暗箱内部的光线。在针孔照相机模型中，在光线投影到图像平面之前，从唯一一个点经过，也就是照相机中心 C。图 4-1 为从照相机中心前画出图像平面的图解。事实上，在真实的照相机中，图像平面位于照相机中心之后，但是照相机的模型和图 4-1 的模型是一样的。

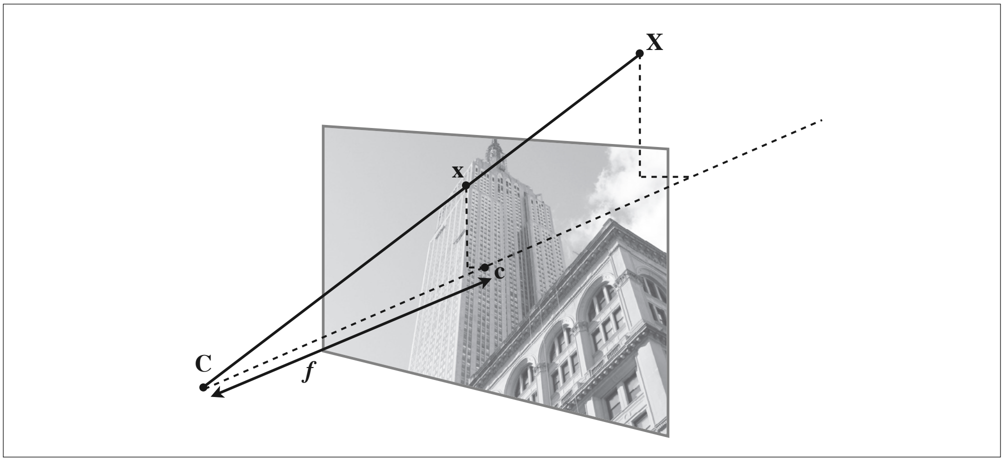


图 4-1：针孔照相机模型。图像点 **x** 是由图像平面与连接三维点 **X** 和照相机中心 **C** 的直线相交而成的。虚线表示该照相机的光学坐标轴

由图像坐标轴和三维坐标系中的 x 轴和 y 轴对齐平行的假设，我们可以得出针孔照相机的投影性质。照相机的光学坐标轴和 z 轴一致，该投影几何可以简化成相似三角形。在投影之前通过旋转和平移变换，对该坐标系加入三维点，会出现完整的投影变换。感兴趣的读者可以查阅文献 [13]、[25] 和 [26]。

在针孔照相机中，三维点 **X** 投影为图像点 **x**（两个点都是用齐次坐标表示的），如下所示：

$$\lambda\mathbf{x} = \boldsymbol{P}\mathbf{X} \tag{4.1}$$

这里，3×4 的矩阵 $\boldsymbol{P}$ 为照相机矩阵（或投影矩阵）。注意，在齐次坐标系中，三维点 **X** 的坐标由 4 个元素组成，$\mathbf{X}=[X, Y, Z, W]$。这里的标量 λ 是三维点的逆深度。如果我们打算在齐次坐标中将最后一个数值归一化为 1，那么就会使用到它。

4.1.1 照相机矩阵

照相机矩阵可以分解为：

$$\boldsymbol{P} = K[\boldsymbol{R} \mid \boldsymbol{t}] \tag{4.2}$$

其中，$\boldsymbol{R}$ 是描述照相机方向的旋转矩阵，$\boldsymbol{t}$ 是描述照相机中心位置的三维平移向量，内标定矩阵 $\boldsymbol{K}$ 描述照相机的投影性质。

标定矩阵仅和照相机自身的情况相关，通常情况下可以写成：

$$\boldsymbol{K}=\begin{bmatrix}\alpha f & s & c_x\\ 0 & f & c_y\\ 0 & 0 & 1\end{bmatrix}$$

图像平面和照相机中心间的距离为焦距 f。当像素数组在传感器上偏斜的时候，需要用到倾斜参数 s。在大多数情况下，s 可以设置成 0。也就是说：

$$\boldsymbol{K}=\begin{bmatrix}f_x & 0 & c_x\\ 0 & f_y & c_y\\ 0 & 0 & 1\end{bmatrix} \tag{4.3}$$

这里，我们使用了另外的记号 f_x 和 f_y，两者关系为 $f_x=\alpha f_y$。

纵横比例参数 α 是在像素元素非正方形的情况下使用的。通常情况下，我们可以默认设置 $\alpha=1$。经过这些假设，标定矩阵变为：

$$\boldsymbol{K}=\begin{bmatrix}f & 0 & c_x\\ 0 & f & c_y\\ 0 & 0 & 1\end{bmatrix}$$

除焦距之外，标定矩阵中剩余的唯一参数为光心（有时称主点）的坐标 $\mathbf{c}=[c_x, c_y]$，也就是光线坐标轴和图像平面的交点。因为光心通常在图像的中心，并且图像的坐标是从左上角开始计算的，所以光心的坐标常接近于图像宽度和高度的一半。特别强调一点，在这个例子中，唯一未知的变量是焦距 f。

4.1.2　三维点的投影

下面来创建照相机类，用来处理我们对照相机和投影建模所需要的全部操作：

```
from scipy import linalg

class Camera(object):
  """ 表示针孔照相机的类 """

  def __init__(self,P):
    """ 初始化 P = K[R|t] 照相机模型 """
    self.P = P
    self.K = None # 标定矩阵
    self.R = None # 旋转
    self.t = None # 平移
    self.c = None # 照相机中心

  def project(self,X):
    """ X（4×n 的数组）的投影点，并且进行坐标归一化 """
```

```
    x = dot(self.P,X)
    for i in range(3):
      x[i] /= x[2]
    return x
```

下面的例子展示如何将三维中的点投影到图像视图中。在这个例子中，我们将使用牛津多视图数据集中的“Model Housing”数据集，可以从 http://www.robots.ox.ac.uk/~vgg/data/data-mview.html 下载。下载这些三维几何图像文件，然后将 house.p3d 文件复制到你的工作目录里：

```
import camera

# 载入点
points = loadtxt('house.p3d').T
points = vstack((points,ones(points.shape[1])))

# 设置照相机参数
P = hstack((eye(3),array([[0],[0],[-10]])))
cam = camera.Camera(P)
x = cam.project(points)

# 绘制投影
figure()
plot(x[0],x[1],'k.')
show()
```

首先，我们使用齐次坐标来表示这些点。然后我们使用一个投影矩阵来创建 Camera 对象将这些三维点投影到图像平面并执行绘制操作，输出结果如图 4-2 中间图像所示。

为了研究照相机的移动会如何改变投影的效果，你可以使用下面的代码。该代码围绕一个随机的三维向量，进行增量旋转的投影。

```
# 创建变换
r = 0.05*random.rand(3)
rot = camera.rotation_matrix(r)

# 旋转矩阵和投影
figure()
for t in range(20):
  cam.P = dot(cam.P,rot)
  x = cam.project(points)
  plot(x[0],x[1],'k.')
```

```
show()
```

在上面的代码中，我们使用了 rotation_matrix() 函数，该函数能够创建围绕一个向量进行三维旋转的旋转矩阵（将该函数添加到 camera.py 文件中）：

```
def rotation_matrix(a):
  """ 创建一个用于围绕向量 a 轴旋转的三维旋转矩阵 """
  R = eye(4)
  R[:3,:3] = linalg.expm([[0,-a[2],a[1]],[a[2],0,-a[0]],[-a[1],a[0],0]])
  return R
```

图 4-2 所示分别为序列中的一幅图像、三维点的投影，以及这些点围绕一个随机向量旋转后三维点投影的轨迹。运行该代码几次，进行不同的随机旋转之后，你会对点在投影中如何旋转有一些感觉。

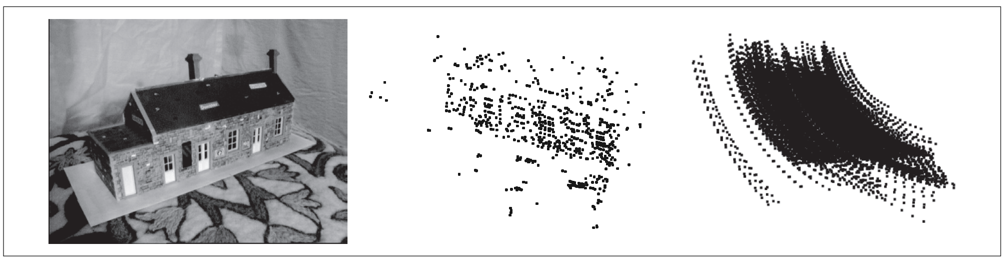


图 4-2：投影三维点示例：样本图像（左）；图像视图中投影后的点（中）；经过照相机旋转后，投影点的轨迹（右）。数据来自于牛津“Model House”数据集

4.1.3　照相机矩阵的分解

如果给定如方程（4.2）所示的照相机矩阵 $\boldsymbol{P}$，我们需要恢复内参数 K 以及照相机的位置 $\mathbf{t}$ 和姿势 $\boldsymbol{R}$。矩阵分块操作称为因子分解。这里，我们将使用一种矩阵因子分解的方法，称为 RQ 因子分解。

将下面的方法添加到 Camera 类中：

```
def factor(self):
  """ 将照相机矩阵分解为 K、R、t，其中，P = K[R|t] """

  # 分解前 3×3 的部分
  K,R = linalg.rq(self.P[:,:3])

  # 将 K 的对角线元素设为正值
  T = diag(sign(diag(K)))
  if linalg.det(T) < 0:
    T[1,1] *= -1
```

```
  self.K = dot(K,T)
  self.R = dot(T,R) # T 的逆矩阵为其自身
  self.t = dot(linalg.inv(self.K),self.P[:,3])

  return self.K, self.R, self.t
```

RQ 因子分解的结果并不是唯一的。在该因子分解中，分解的结果存在符号二义性。由于我们需要限制旋转矩阵 $\boldsymbol{R}$ 为正定的（否则，旋转坐标轴即可），所以如果需要，我们可以在求解到的结果中加入变换 $\boldsymbol{T}$ 来改变符号。

在示例照相机上运行下面的代码，观察照相机矩阵分解的效果：

```
import camera

K = array([[1000,0,500],[0,1000,300],[0,0,1]])
tmp = camera.rotation_matrix([0,0,1])[:3,:3]
Rt = hstack((tmp,array([[50],[40],[30]])))
cam = camera.Camera(dot(K,Rt))

print K,Rt
print cam.factor()
```

你会在控制台上得到相同的输出。

4.1.4 计算照相机中心

给定照相机投影矩阵 $\boldsymbol{P}$，我们可以计算出空间上照相机的所在位置。照相机的中心 $\mathbf{C}$，是一个三维点，满足约束 $\boldsymbol{P}\mathbf{C}=0$。对于投影矩阵为 $\boldsymbol{P}=K[\boldsymbol{R}|\boldsymbol{t}]$ 的照相机，有：

$$\boldsymbol{K}[\boldsymbol{R}\,|\,\boldsymbol{t}]\mathbf{C} = \boldsymbol{K}\ \boldsymbol{R}\mathbf{C} + \boldsymbol{K}\boldsymbol{t} = 0$$

照相机的中心可以由下述式子来计算：

$$\mathbf{C} = -\,\boldsymbol{R}^T\boldsymbol{t}$$

注意，如预期一样，照相机的中心和内标定矩阵 $\boldsymbol{K}$ 无关。

下面的代码可以按照上面公式计算照相机的中心。将其添加到 Camera 类中，该方法会返回照相机的中心：

```
def center(self):
  """ 计算并返回照相机的中心 """

  if self.c is not None:
```

```
        return self.c
    else:
        # 通过因子分解计算 c
        self.factor()
        self.c = -dot(self.R.T,self.t)
        return self.c
```

上面的一些方法构成了 Camera 类的基本函数操作。现在，让我们一起探讨如何使用针孔照相机模型。

4.2 照相机标定

标定照相机是指计算出该照相机的内参数。在我们的例子中，是指计算矩阵 $\boldsymbol{K}$。如果你的应用要求高精度，那么可以扩展该照相机模型，使其包含径向畸变和其他条件。对于大多数应用来说，公式（4.3）中的简单照相机模型已经足够。标定照相机的标准方法是，拍摄多幅平面棋盘模式的图像，然后进行处理计算。例如，OpenCV 中的标定工具使用了这种方法（详见文献 [3]）。

4.2.1 一个简单的标定方法

这里我们将要介绍一个简单的照相机标定方法。大多数参数可以使用基本的假设来设定（正方形垂直的像素，光心位于图像中心），比较难处理的是获得正确的焦距。对于这种标定方法，你需要准备一个平面矩形的标定物体（一个书本即可）、用于测量的卷尺和直尺，以及一个平面。下面是具体操作步骤：

- 测量你选定矩形标定物体的边长 $\mathrm{d}X$ 和 $\mathrm{d}Y$；
- 将照相机和标定物体放置在平面上，使得照相机的背面和标定物体平行，同时物体位于照相机图像视图的中心，你可能需要调整照相机或者物体来获得良好的对齐效果；
- 测量标定物体到照相机的距离 $\mathrm{d}Z$；
- 拍摄一副图像来检验该设置是否正确，即标定物体的边要和图像的行和列对齐；
- 使用像素数来测量标定物体图像的宽度和高度 $\mathrm{d}x$ 和 $\mathrm{d}y$。

实验设置情况如图 4-3 所示。现在，使用下面的相似三角形（参见图 4-1）关系可以获得焦距：

$$f_x = \frac{\mathrm{d}x}{\mathrm{d}X}\mathrm{d}Z, \quad f_y = \frac{\mathrm{d}y}{\mathrm{d}Y}\mathrm{d}Z$$

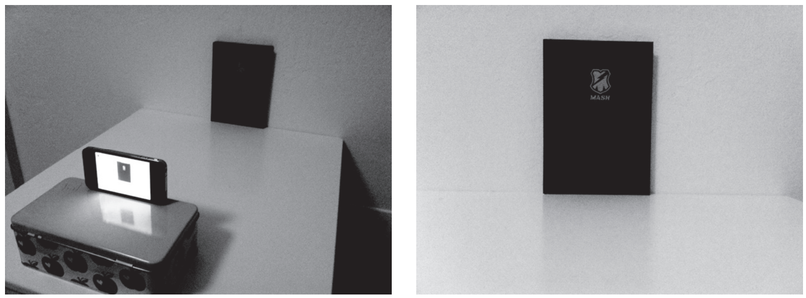


图 4-3：简单的照相机标定设置：进行标定实验使用的设置情况图像（左）；用于标定的图像（右）。在图像中测量标定物体的宽度和高度，以及设置中标定物体的实际物理尺寸，就可以确定焦距的大小

对于如图 4-3 所示的特定设置，物体宽度和高度的测量值分别为 130 mm 和 185 mm，则，dX=130，dY=185。从照相机到物体的距离为 460 mm，则 dZ=460。你可以使用任意的测量单位，只有测量值的比例才影响最终焦距的计算。你可以使用 `ginput()` 函数来获得图像中的 4 个点，图像中物体的宽度和高度分别为 722 和 1040 像素。将这些值代入上面的关系表达式可以获得焦距的大小：

$$f_x = 2555,\quad f_y = 2586$$

值得注意的是，我们现在获取的焦距是在特定图像分辨率下计算出来的。在这个例子中，图像大小为 2592 × 1936 像素。记住，焦距和光心是使用像素来度量的，其尺度和图像分辨率相关。如果你使用其他的图像分辨率来拍摄（例如，一个缩略图像），那么这些值都会改变。将照相机的这些测量数值写入到一个如下所示的辅助函数中，会方便一些：

```
def my_calibration(sz):
  row,col = sz
  fx = 2555*col/2592
  fy = 2586*row/1936
  K = diag([fx,fy,1])
  K[0,2] = 0.5*col
  K[1,2] = 0.5*row
  return K
```

该函数的输入参数为表示图像大小的元组，返回参数为标定矩阵。这里，我们假设光心为图像的中心。如果你喜欢的话，可以按照焦距的意义来替换焦距的值；对于大多数普通照相机来说，这样做是可以的。注意，这里的标定是对于横向旋转的图像。对于纵向旋转，你需要交换标定矩阵中焦距的值。我们会保留上面的这个函数，方便在下面的章节中使用。

4.3 以平面和标记物进行姿态估计

在第 3 章中，我们学习了如何从平面间估计单应性矩阵。如果图像中包含平面状的标记物体，并且已经对照相机进行了标定，那么我们可以计算出照相机的姿态（旋转和平移）。这里的标记物体可以为对任何平坦的物体。

我们使用一个例子来演示如何进行姿态估计。这里借助图 4-4 中顶端的两幅图像。我们使用下面的代码来提取两幅图像的 SIFT 特征，然后使用 RANSAC 算法稳健地估计单应性矩阵：

```
import homography
import camera
import sift

# 计算特征
sift.process_image('book_frontal.JPG','im0.sift')
l0,d0 = sift.read_features_from_file('im0.sift')

sift.process_image('book_perspective.JPG','im1.sift')
l1,d1 = sift.read_features_from_file('im1.sift')

# 匹配特征，并计算单应性矩阵
matches = sift.match_twosided(d0,d1)
ndx = matches.nonzero()[0]
fp = homography.make_homog(l0[ndx,:2].T)
ndx2 = [int(matches[i]) for i in ndx]
tp = homography.make_homog(l1[ndx2,:2].T)

model = homography.RansacModel()
H = homography.H_from_ransac(fp,tp,model)
```

现在我们得到了单应性矩阵。该单应性矩阵将一幅图像中标记物（在这个例子中，标记物是指书本）上的点映射到另一幅图像中的对应点。下面我们定义相应的三维坐标系，使标记物在 *X*-*Y* 平面上（*Z*=0），原点在标记物的某位置上。

为了检验单应性矩阵结果的正确性，我们需要将一些简单的三维物体放置在标记物上，这里我们使用一个立方体。你可以使用下面的函数来产生立方体上的点：

```
def cube_points(c,wid):
  """ 创建用于绘制立方体的一个点列表（前 5 个点是底部的正方形，一些边重合了）"""
  p = []
  # 底部
  p.append([c[0]-wid,c[1]-wid,c[2]-wid])
  p.append([c[0]-wid,c[1]+wid,c[2]-wid])
```

```
    p.append([c[0]+wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]-wid,c[2]-wid])
    p.append([c[0]-wid,c[1]-wid,c[2]-wid]) # 为了绘制闭合图像，和第一个相同

    # 顶部
    p.append([c[0]-wid,c[1]-wid,c[2]+wid])
    p.append([c[0]-wid,c[1]+wid,c[2]+wid])
    p.append([c[0]+wid,c[1]+wid,c[2]+wid])
    p.append([c[0]+wid,c[1]-wid,c[2]+wid])
    p.append([c[0]-wid,c[1]-wid,c[2]+wid]) # 为了绘制闭合图像，和第一个相同

    # 竖直边
    p.append([c[0]-wid,c[1]-wid,c[2]+wid])
    p.append([c[0]-wid,c[1]+wid,c[2]+wid])
    p.append([c[0]-wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]+wid,c[2]+wid])
    p.append([c[0]+wid,c[1]-wid,c[2]+wid])
    p.append([c[0]+wid,c[1]-wid,c[2]-wid])

    return array(p).T
```

在上面的函数中，一些数据点会重复出现，plot() 函数会绘制出漂亮的立方体。



图 4-4：使用平面物体作为标记物，来计算用于新视图投影矩阵的例子。将图像的特征和对齐后的标记匹配，计算出单应性矩阵，然后用于计算照相机的姿态。带有一个灰色正方形区域的模板图像（左上）；从未知视角拍摄的一幅图像，该图像包含同一个正方形，该正方形已经经过估计的单应性矩阵进行了变换（右上）；使用计算出的照相机矩阵变换立方体（下）

有了单应性矩阵和照相机的标定矩阵，我们现在可以得出两个视图间的相对变换：

```
# 计算照相机标定矩阵
K = my_calibration((747,1000))

# 位于边长为 0.2，z=0 平面上的三维点
box = cube_points([0,0,0.1],0.1)

# 投影第一幅图像上底部的正方形
cam1 = camera.Camera( hstack((K,dot(K,array([[0],[0],[-1]])) )) )
# 底部正方形上的点
box_cam1 = cam1.project(homography.make_homog(box[:,:5]))

# 使用 H 将点变换到第二幅图像中
box_trans = homography.normalize(dot(H,box_cam1))

# 从 cam1 和 H 中计算第二个照相机矩阵
cam2 = camera.Camera(dot(H,cam1.P))
A = dot(linalg.inv(K),cam2.P[:,:3])
A = array([A[:,0],A[:,1],cross(A[:,0],A[:,1])]).T
cam2.P[:,:3] = dot(K,A)

# 使用第二个照相机矩阵投影
box_cam2 = cam2.project(homography.make_homog(box))

# 测试：将点投影在 z=0 上，应该能够得到相同的点
point = array([1,1,0,1]).T
print homography.normalize(dot(dot(H,cam1.P),point))
print cam2.project(point)
```

这里我们使用图像的分辨率为 747×1000，第一个产生的标定矩阵就是在该图像分辨率大小下的标定矩阵。下面，我们在原点附近创建立方体上的点。cube_points() 函数产生的前五个点对应于立方体底部的点，在这个例子中对应于位于标记物上 Z=0 平面内的点。第一幅图像（图 4-4 左上）是书本的主视图，我们将其作为这个例子中的模板图像。因为场景坐标的尺度是任意的，所以我们使用下面的矩阵来创建第一个照相机：

$$\boldsymbol{P}_1 = \boldsymbol{K}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix}$$

其中，图像的坐标轴和照相机是对齐的，并且放置在标记物的正上方。将前 5 个三维点投影到图像上。有了估计出的单应性矩阵，我们可以将其变换到第二幅图像上。绘

制出变换后的图像，并在同样的标记物位置绘制出这些点（如图 4-4 右上所示）。

现在，结合 $\boldsymbol{P}_1$ 和 $\boldsymbol{H}$ 构建第二幅图像的照相机矩阵：

$$\boldsymbol{P}_2=\boldsymbol{H}\boldsymbol{P}_1$$

该矩阵将标记平面 Z=0 上的点变换到正确的位置。也就是说，$\boldsymbol{P}_2$ 矩阵的前两列和第四列是正确的。由于我们知道前 3×3 矩阵块应该为 $\boldsymbol{KR}$，并且 $\boldsymbol{R}$ 是旋转矩阵，所以我们可以将 $\boldsymbol{P}_2$ 乘以标定矩阵的逆，然后将第三列替换为前两列的交叉乘积，以此来恢复第三列。

作为合理性验证，我们可以使用新矩阵投影标记平面的一个点，然后检查投影后的点是否与使用第一个照相机和单应性矩阵变换后的点相同。你会发现，控制台上得到了相同的输出结果。

你可以用如下所示的代码来可视化这些投影后的点：

```
im0 = array(Image.open('book_frontal.JPG'))
im1 = array(Image.open('book_perspective.JPG'))

# 底部正方形的二维投影
figure()
imshow(im0)
plot(box_cam1[0,:],box_cam1[1,:],linewidth=3)

# 使用 H 对二维投影进行变换
figure()
imshow(im1)
plot(box_trans[0,:],box_trans[1,:],linewidth=3)

# 三维立方体
figure()
imshow(im1)
plot(box_cam2[0,:],box_cam2[1,:],linewidth=3)

show()
```

该代码绘制出如图 4-4 所示的三幅图像。为了能够在后面的例子中再次使用这些计算的结果，我们可以使用 Pickle 将这些照相机矩阵保存起来：

```
import pickle

with open('ar_camera.pkl','w') as f:
  pickle.dump(K,f)
  pickle.dump(dot(linalg.inv(K),cam2.P),f)
```

现在我们已经学会计算给定平面场景物体的照相机矩阵。我们结合特征匹配和单应性矩阵，以及照相机标定技术，简单演示了如何在一幅图像上放置一个立方体。有了照相机的姿态估计技术，我们现在就具备了创建简单增强现实应用的基本技能了。

4.4 增强现实

增强现实（Augmented Reality，AR）是将物体和相应信息放置在图像数据上的一系列操作的总称。最经典的例子是放置一个三维计算机图形学模型，使其看起来属于该场景；如果在视频中，该模型会随着照相机的运动很自然地移动。如上一节所示，给定一幅带有标记平面的图像，我们能够计算出照相机的位置和姿态，使用这些信息来放置计算机图形学模型，能够正确表示它们。在本章的最后一节，我们将介绍如何建立一个简单的增强现实例子。其中，我们会用到两个工具包：PyGame 和 PyOpenGL。

4.4.1 PyGame和PyOpenGL

PyGame 是非常流行的游戏开发工具包，它可以非常简单地处理显示窗口、输入设备、事件，以及其他内容。PyGame 是开源的，可以从 http://www.pygame.org/ 下载。事实上，它是一个 Python 绑定的 SDL 游戏引擎。你可以查看附录 A 来获取关于安装的帮助。你还可以查看文献 [21] 来获取关于 PyGame 工具包的更多编程细节。

PyOpenGL 是 OpenGL 图形编程的 Python 绑定接口。OpenGL 可以安装在几乎所有的系统上，并且具有很好的图形性能。OpenGL 具有跨平台性，能够在不同的操作系统之间工作。关于 OpenGL 的更多信息，参见 http://www.opengl.org/。在开始页面（http://www.opengl.org/wiki/Getting_started），有针对初学者的很多资源。PyOpenGL 是开源的，并且易于安装。你可以从附录 A 了解关于 PyOpenGL 更多内容。你同样可以在项目网页 http://pyopengl.sourceforge.net/ 获取更多细节内容。

我们没有涵盖关于 OpenGL 编程的绝大部分内容。相反，我们在这里只讲述重要的内容，例如如何在 OpenGL 中使用照相机矩阵，如何设置一个基本的三维模型。你可以从 PyOpenGL-Demo 工具包（http://pypi.python.org/pypi/PyOpenGL-Demo）获取一些很好的例子和演示。如果你不熟悉 PyOpenGL，那么该工具包是个很好的开始。

我们使用 OpenGL 将一个三维模型放置在一个场景中。为了使用 PyGame 和 PyOpenGL 工具包来完成该应用，需要在脚本的开始部分载入下面的命令：

```
from OpenGL.GL import *
from OpenGL.GLU import *
import pygame, pygame.image
from pygame.locals import *
```

你可以看到，这里主要使用 OpenGL 中的两个部分：GL 部分包含所有以“gl”开头的函数，其中包含我们需要的大部分函数；GLU 部分是 OpenGL 的实用函数库，里面包含了一些高层的函数。我们主要使用它来设置照相机投影。pygame 部分用来设置窗口和事件控制；其中 pygame.image 用来载入图像和创建 OpenGL 的纹理，pygame.locals 用来设置 OpenGL 的显示区域。

需要对一个 OpenGL 场景进行两个部分的设置：投影和视图矩阵的建模。让我们一起来学习如何由针孔照相机来创建这些矩阵。

4.4.2 从照相机矩阵到OpenGL格式

OpenGL 使用 4×4 的矩阵来表示变换（包括三维变换和投影）。这和我们使用的 3×4 照相机矩阵略有差别。但是，照相机与场景的变换分成了两个矩阵，GL_PROJECTION 矩阵和 GL_MODELVIEW 矩阵。GL_PROJECTION 矩阵处理图像成像的性质，等价于我们的内标定矩阵 $\boldsymbol{K}$。GL_MODELVIEW 矩阵处理物体和照相机之间的三维变换关系，对应于我们照相机矩阵中的 $\boldsymbol{R}$ 和 $\boldsymbol{t}$ 部分。一个不同之处是，假设照相机为坐标系的中心，GL_MODELVIEW 矩阵实际上包含了将物体放置在照相机前面的变换。OpenGL 有很多特性，我们会在下面例子中一一讲解。

假设我们已经获得了标定好的照相机，即已知标定矩阵 $\boldsymbol{K}$，下面的函数可以将照相机参数转换为 OpenGL 中的投影矩阵：

```
def set_projection_from_camera(K):
  """ 从照相机标定矩阵中获得视图 """

  glMatrixMode(GL_PROJECTION)
  glLoadIdentity()

  fx = K[0,0]
  fy = K[1,1]
  fovy = 2*arctan(0.5*height/fy)*180/pi
  aspect = (width*fy)/(height*fx)

  # 定义近的和远的剪裁平面
  near = 0.1
  far = 100.0
```

```
# 设定透视
gluPerspective(fovy,aspect,near,far)
glViewport(0,0,width,height)
```

我们假设标定矩阵如（4.3）那样简单，光心为图像的中心。第一个函数 glMatrixMode() 将工作矩阵设置为 GL_PROJECTION，接下来的命令会修改这个矩阵[1]。然后，glLoadIdentity() 函数将该矩阵设置为单位矩阵，这是重置矩阵的一般操作。然后，我们根据图像的高度、照相机的焦距以及纵横比，计算出视图中的垂直场。OpenGL 的投影同样具有近距离和远距离的裁剪平面来限制场景拍摄的深度范围。我们设置近深度为一个小的数值，使得照相机能够包含最近的物体，而远深度设置为一个大的数值。我们使用 GLU 的实用函数 gluPerspective() 来设置投影矩阵，将整个图像定义为视图部分（也就是显示的部分）。和下面的模拟视图函数相似，你可以使用 glLoadMatrixf() 函数的一个选项来定义一个完全的投影矩阵。当简单版本的标定矩阵不够好时，可以使用完全投影矩阵。

模拟视图矩阵能够表示相对的旋转和平移，该变换将该物体放置在照相机前（效果是照相机在原点上）。模拟视图矩阵是个典型的 4 × 4 矩阵，如下所示：

$$\begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix}$$

其中，$\boldsymbol{R}$ 是旋转矩阵，列向量表示 3 个坐标轴的方向，$\boldsymbol{t}$ 是平移向量。当创建模拟视图矩阵时，旋转矩阵需要包括所有的旋转（物体和坐标系的旋转），可以将单个旋转分量相乘来获得旋转矩阵。

下面的函数实现如何获得移除标定矩阵后的 3 × 4 针孔照相机矩阵（将 $\boldsymbol{P}$ 和 $\boldsymbol{K}^{-1}$ 相乘），并创建一个模拟视图：

```
def set_modelview_from_camera(Rt):
  """ 从照相机姿态中获得模拟视图矩阵 """

  glMatrixMode(GL_MODELVIEW)
  glLoadIdentity()

  # 围绕 x 轴将茶壶旋转 90 度，使 z 轴向上
  Rx = array([[1,0,0],[0,0,-1],[0,1,0]])

  # 获得旋转的最佳逼近
```

注 1：这是个奇怪的处理方式，但是只需要处理 GL_PROJECTION 和 GL_MODELVIEW 两个矩阵，所以是可行的。

```
    R = Rt[:,:3]
    U,S,V = linalg.svd(R)
    R = dot(U,V)
    R[0,:] = -R[0,:] # 改变 x 轴的符号

    # 获得平移量
    t = Rt[:,3]

    # 获得 4×4 的模拟视图矩阵
    M = eye(4)
    M[:3,:3] = dot(R,Rx)
    M[:3,3] = t

    # 转置并压平以获取列序数值
    M = M.T
    m = M.flatten()

    # 将模拟视图矩阵替换为新的矩阵
    glLoadMatrixf(m)
```

在上面函数中，我们首先切换到 GL_MODELVIEW 矩阵，然后重置该矩阵。然后，由于需要旋转该物体（你将在下面看到），所以我们创建一个 90 度的旋转矩阵。接下来，由于估计照相机矩阵时，可能会有错误或者噪声干扰，所以我们确保照相机矩阵的旋转部分确实是个旋转矩阵。该操作使用 SVD 分解方法，旋转矩阵的最佳逼近可以通过 $\boldsymbol{R}=UV^T$ 来获得。由于 OpenGL 中的坐标系和上面用到的有点不同，所以我们将 x 轴翻转。然后，我们将模拟视图矩阵 $\boldsymbol{M}$ 设置为旋转矩阵的乘积。`glLoadMatrixf()` 函数通过输入参数为按列排列的 16 个数值数组，来设置模拟视图。将 $\boldsymbol{M}$ 矩阵转置，然后压平并输入 `glLoadMatrixf()` 函数。

4.4.3 在图像中放置虚拟物体

我们需要做的第一件事是将图像（打算放置虚拟物体的图像）作为背景添加进来。在 OpenGL 中，该操作可以通过创建一个四边形的方式来完成，该四边形为整个视图。完成该操作最简单的方式是绘制出四边形，同时将投影和模拟试图矩阵重置，使得每一维的坐标范围在 −1 到 1 之间。

下面的函数可以载入一幅图像，然后将其转换成一个 OpenGL 纹理，并将该纹理放置在四边形上：

```
def draw_background(imname):
  """ 使用四边形绘制背景图像 """
```

```
# 载入背景图像（应该是 .bmp 格式），转换为 OpenGL 纹理
bg_image = pygame.image.load(imname).convert()
bg_data = pygame.image.tostring(bg_image,"RGBX",1)

glMatrixMode(GL_MODELVIEW)
glLoadIdentity()
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

# 绑定纹理
glEnable(GL_TEXTURE_2D)
glBindTexture(GL_TEXTURE_2D,glGenTextures(1))
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,width,height,0,GL_RGBA,GL_UNSIGNED_BYTE,bg_data)
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST)
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST)

# 创建四方形填充整个窗口
glBegin(GL_QUADS)
glTexCoord2f(0.0,0.0); glVertex3f(-1.0,-1.0,-1.0)
glTexCoord2f(1.0,0.0); glVertex3f( 1.0,-1.0,-1.0)
glTexCoord2f(1.0,1.0); glVertex3f( 1.0, 1.0,-1.0)
glTexCoord2f(0.0,1.0); glVertex3f(-1.0, 1.0,-1.0)
glEnd()

# 清除纹理
glDeleteTextures(1)
```

该函数首先使用 PyGame 中的一些函数来载入一幅图像，将其序列化为能够在 PyOpenGL 中使用的原始字符串表示。然后，重置模拟视图，清除颜色和深度缓存。接下来，绑定这个纹理，使其能够在四边形和指定插值中使用它。四边形是在每一维分别为 −1 和 1 的点上定义的。注意，纹理图像的坐标是从 0 到 1。最后，清除该纹理，避免其干扰之后准备绘制的图像。

现在已经准备好将物体放置入场景中。我们将使用“hello world”的计算机图形学例子，Utah 茶壶（http://en.wikipedia.org/wiki/Utah_teapot）。这个茶壶有丰富的历史，在 GLUT 用作一个标准形状：

```
from OpenGL.GLUT import *
glutSolidTeapot(size)
```

该命令产生一个相对大小为 `size` 的茶壶模型。

下面的函数将会设置颜色和其他特性，产生一个红色的漂亮茶壶：

```
def draw_teapot(size):
  """ 在原点处绘制红色茶壶 """
```

```
glEnable(GL_LIGHTING)
glEnable(GL_LIGHT0)
glEnable(GL_DEPTH_TEST)
glClear(GL_DEPTH_BUFFER_BIT)

# 绘制红色茶壶
glMaterialfv(GL_FRONT,GL_AMBIENT,[0,0,0,0])
glMaterialfv(GL_FRONT,GL_DIFFUSE,[0.5,0.0,0.0,0.0])
glMaterialfv(GL_FRONT,GL_SPECULAR,[0.7,0.6,0.6,0.0])
glMaterialf(GL_FRONT,GL_SHININESS,0.25*128.0)
glutSolidTeapot(size)
```

上面的函数中，前两行激活了灯光效果和一盏灯。灯被计为 GL_LIGHT0 和 GL_LIGHT1 等。在本例中，我们只使用一盏灯。`glEnable()` 函数用来激活 OpenGL 的一些特性。这些特性是使用大写常量来定义的。关闭特性可以使用相应的 `glDisable()` 函数来完成。接下来，深度测试被激活，使物体按照其深度表示出来（远处的物体不能绘制在近处物体的前面），然后清理深度缓存。接下来，指定物体的物质特性，例如漫反射和镜面反射颜色。在最后一行代码中，将指定的物质特性加入到 Utah 茶壶上。

4.4.4 综合集成

下面的完整脚本可以生成如图 4-5 所示的图像（假设你已经将上面的函数添加到了同一个文件中）：

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import pygame, pygame.image
from pygame.locals import *
import pickle

width,height = 1000,747

def setup():
  """ 设置窗口和 pygame 环境 """
  pygame.init()
  pygame.display.set_mode((width,height),OPENGL | DOUBLEBUF)
  pygame.display.set_caption('OpenGL AR demo')

# 载入照相机数据
with open('ar_camera.pkl','r') as f:
  K = pickle.load(f)
  Rt = pickle.load(f)
```

```
setup()
draw_background('book_perspective.bmp')
set_projection_from_camera(K)
set_modelview_from_camera(Rt)
draw_teapot(0.02)

while True:
  event = pygame.event.poll()
  if event.type in (QUIT,KEYDOWN):
  break
pygame.display.flip()
```

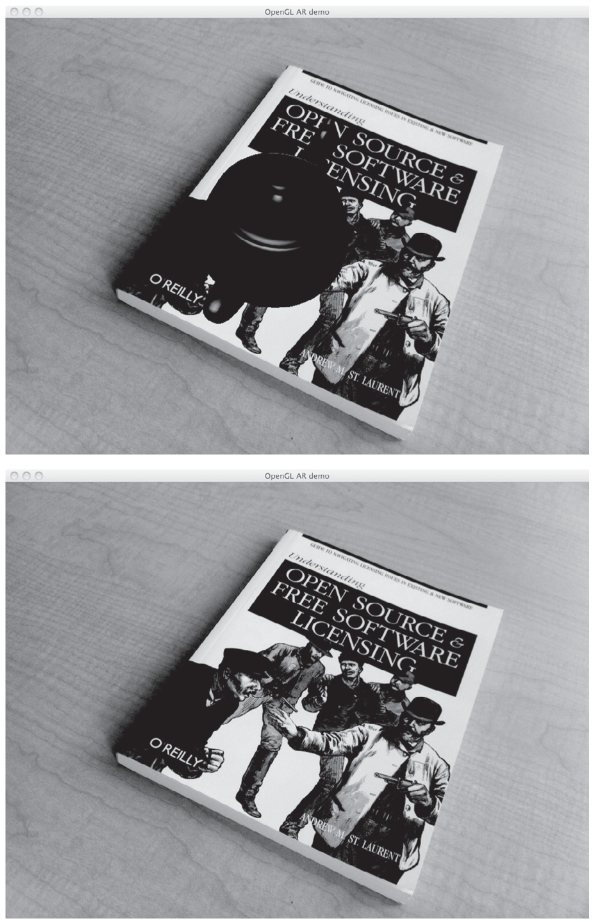


图 4-5：增强现实。使用由特征匹配计算出的照相机参数，将一个计算机图形学模型放置在场景中的书本上：将 Utah 茶壶按照和坐标轴对齐的方式显示出来（上）；进行合理性验证，查看原点的位置（下）

首先，该脚本使用 Pickle 载入照相机标定矩阵，以及照相机矩阵中的旋转和平移部分。假设这些信息已经按照 4.3 节的描述进行保存。`setup()` 函数会初始化 PyGame，将窗口设置为图像的大小，绘制图像区域为两倍的 OpenGL 窗口缓存大小。接下来，载入背景图像，使其与窗口相符。然后，设定照相机和模拟视图矩阵。最后，在正确的位置上绘制出茶壶。

在 PyGame 中，使用带有对所有变化进行定期询问的无限循环来处理事件。这些事件可以为键盘、鼠标，或者其他。在这个例子中，我们检查应用是否退出，或者是否有按键按下，如果是，则退出这个循环。`pygame.display.flip()` 命令将会在屏幕上绘制出物体。

程序的输出结果如图 4-5 所示。可以看到，物体的朝向是正确的（在图 4-4 中，茶壶和立方体的边是对齐的）。为了验证放置的正确性，你可以通过给 `size` 变量传递一个较小的数值，将茶壶变小。在图 4-4 中，茶壶应该放置在靠近立方体坐标为 [0, 0, 0] 的角上。图 4-5 给出了一个例子。

4.4.5　载入模型

在结束本章之前，我们讲述最后一个细节：载入并显示三维模型。你可以在 http://www.pygame.org/wiki/OBJFileLoader 了解如何在 PyGame 中载入保存在 .obj 格式文件中的模型。除此之外，你还可以在 http://en.wikipedia.org/wiki/Wavefront_.obj_file 学习关于 .obj 以及其他相关文件格式的更多内容。

下面使用一个基本的例子来说明其使用方法。我们将使用一个免费的玩具飞机模型，模型来自 http://www.oyonale.com/modeles.php[1]。下载其 .obj 文件，然后保存为 toyplane.obj。当然，你也可以用任何其他模型替换，下面的代码不变。

假设已经下载模型文件并命名为 objloader.py, 将下面的函数添加到茶壶例子的代码文件中：

```
def load_and_draw_model(filename):
  """ 使用 objloader.py，从 .obj 文件中装载模型
    假设路径文件夹中存在同名的 .mtl 材料设置文件 """
  glEnable(GL_LIGHTING)
  glEnable(GL_LIGHT0)
  glEnable(GL_DEPTH_TEST)
  glClear(GL_DEPTH_BUFFER_BIT)

  # 设置模型颜色
```

注 1：模型由 Gilles Tran 提供（共享创意许可）。

```
    glMaterialfv(GL_FRONT,GL_AMBIENT,[0,0,0,0])
    glMaterialfv(GL_FRONT,GL_DIFFUSE,[0.5,0.75,1.0,0.0])
    glMaterialf(GL_FRONT,GL_SHININESS,0.25*128.0)

    # 从文件中载入
    import objloader
    obj = objloader.OBJ(filename,swapyz=True)
    glCallList(obj.gl_list)
```

和前面一样，我们首先设置模型的照明条件和颜色属性。接下来，我们将模型载入一个 obj 对象中，然后在文件中执行 OpenGL 的调用。

你可以在相应的 .mtl 文件中设置纹理和材料属性。实际上，objloader 模块需要一个含有材料设置的文件。我们采用仅创建一个小材料文件的实用方法，而不修改脚本。在这个例子中，我们仅指定了颜色信息。

使用下面的命令来创建 toyplane.mtl 文件：

```
newmtl lightblue
Kd 0.5 0.75 1.0
illum 1
```

上面的命令将物体的漫反射颜色设置为浅灰蓝色。现在，确保将 .obj 文件中的"usemtl"标签替换为：

```
usemtl lightblue
```

我们将添加纹理留作练习。将上面例子中的 draw_teapot() 命令替换为：

```
load_and_draw_model('toyplane.obj')
```

程序就会生成如图 4-6 所示的窗口图像。

这是本书关于增强现实和 OpenGL 最深入的例子。学习了标定矩阵、计算照相机姿态、将照相机转变成 OpenGL 格式，以及在场景中显示模型等方法，你已经具备了继续探索增强现实的基础。在下一节中，我们将在不使用标记物的情况下，继续学习照相机模型，计算三维结构以及照相机姿态。

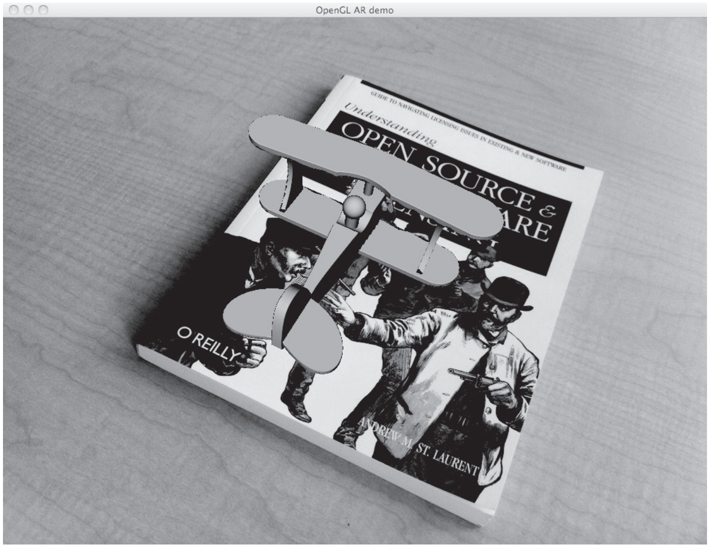


图 4-6：从 .obj 文件载入三维模型，并使用由特征匹配计算出的照相机参数将其放置在书本上

练习

(1) 修改用于图 4-2 运动的示例代码，使其能够对点而非照相机进行变换，你应该会得到相同的图像。使用不同的变换进行实验，然后绘制出相应的结果。

(2) 一些牛津多视图数据集已经给定了照相机矩阵。对一个这样的数据集，计算其照相机位置，并且绘制出照相机的路径。这和你在图像中看到的是否吻合？

(3) 拍摄一些带有平面标记物和物体的场景图像。将图像的特征和一幅全景图像的特征相匹配，计算出每幅图像照相机位置的姿态。绘制出照相机的轨迹，以及标记物的平面。如果你喜欢，也可以在图像中加入特征点。

(4) 在增强现实的例子中，假设物体放置在原点，并且只将照相机的位置应用到模拟视图矩阵中。将物体之间的变换加入到矩阵中来修改该例子，使得在不同的位置放置一些物体。例如，在标记位置处放置茶壶网格。

(5) 浏览 .obj 模型文件的在线文档，学习如何使用纹理模型。找一个模型（或者创建自己的），然后将其添加到场景中。

第 5 章

多视图几何

本章讲解如何处理多个视图，以及如何利用多个视图的几何关系来恢复照相机位置信息和三维结构。通过在不同视点拍摄的图像，我们可以利用特征匹配来计算出三维场景点以及照相机位置。本章会介绍一些基本的方法，展示一个三维重建的完整例子；本章最后将介绍如何由立体图像进行致密深度重建。

5.1 外极几何

多视图几何是利用在不同视点所拍摄图像间的关系，来研究照相机之间或者特征之间关系的一门科学。图像的特征通常是兴趣点，本章使用的也是兴趣点特征。多视图几何中最重要的内容是双视图几何。

如果有一个场景的两个视图以及视图中的对应图像点，那么根据照相机间的空间相对位置关系、照相机的性质以及三维场景点的位置，可以得到对这些图像点的一些几何关系约束。我们通过外极几何来描述这些几何关系。本节简要介绍外极几何的基本内容。关于外极几何的更多内容，参阅文献 [13]。

没有关于照相机的先验知识，会出现固有二义性，因为三维场景点 $\mathbf{X}$ 经过 4×4 的单应性矩阵 $\boldsymbol{H}$ 变换为 $\boldsymbol{H}\mathbf{X}$ 后，则 $\boldsymbol{H}\mathbf{X}$ 在照相机 $\boldsymbol{P}\boldsymbol{H}^{-1}$ 里得到的图像点和 $\mathbf{X}$ 在照相机 P 里得到的图像点相同。利用照相机方程，可以将上述问题描述为：

$$\lambda\mathbf{x} = \boldsymbol{P}\mathbf{X} = \boldsymbol{P}\boldsymbol{H}^{-1}\boldsymbol{H}\mathbf{X} = \hat{\boldsymbol{P}}\hat{\mathbf{X}}$$

因此，当我们分析双视图几何关系时，总是可以将照相机间的相对位置关系用单应

性矩阵加以简化。这里的单应性矩阵通常只做刚体变换，即只通过单应矩阵变换了坐标系。一个比较好的做法是，将原点和坐标轴与第一个照相机对齐，则：

$$\boldsymbol{P}_1 = \boldsymbol{K}_1[\boldsymbol{I} \mid 0] \text{和} \boldsymbol{P}_2 = \boldsymbol{K}_2[\boldsymbol{R} \mid \boldsymbol{t}]$$

为了方便读者阅读，我们在这里采取第 4 章的符号，其中 $\boldsymbol{K}_1$ 和 $\boldsymbol{K}_2$ 是标定矩阵，$\boldsymbol{R}$ 是第二个照相机的旋转矩阵，$\boldsymbol{t}$ 是第二个照相机的平移向量。利用这些照相机参数矩阵，我们可以找到点 $\mathbf{X}$ 的投影点 $\mathbf{x}_1$ 和 $\mathbf{x}_2$（分别对应于投影矩阵 $\boldsymbol{P}_1$ 和 $\boldsymbol{P}_2$）的关系。这样，我们可以从寻找对应的图像出发，恢复照相机参数矩阵。

同一个图像点经过不同的投影矩阵产生的不同投影点必须满足：

$$\mathbf{x}_2^T \boldsymbol{F} \mathbf{x}_1 = 0 \tag{5.1}$$

其中：

$$\boldsymbol{F} = \boldsymbol{K}_2^{-T} \boldsymbol{S}_{\mathrm{t}} \boldsymbol{R}\, \boldsymbol{K}_1^{-1}$$

矩阵 $\boldsymbol{S}_{\mathrm{t}}$ 为反对称矩阵：

$$\boldsymbol{S}_{\mathrm{t}} = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix} \tag{5.2}$$

公式（5.1）为外极约束条件。矩阵 $\boldsymbol{F}$ 为基础矩阵。基础矩阵可以由两照相机的参数矩阵（相对旋转 $\boldsymbol{R}$ 和平移 $\boldsymbol{t}$）表示。由于 $\det(\boldsymbol{F})=0$，所以基础矩阵的秩小于等于 2。我们将在估计 F 的算法中用到这些性质。

上面的公式表明，我们可以借助 $\boldsymbol{F}$ 来恢复出照相机参数，而 $\boldsymbol{F}$ 可以从对应的投影图像点计算出来。在不知道照相机内参数（$\boldsymbol{K}_1$ 和 $\boldsymbol{K}_2$）的情况下，仅能恢复照相机的投影变换矩阵。在知道照相机内参数的情况下，重建是基于度量的。所谓度量重建，即能够在三维重建中正确地表示距离和角度[1]。

利用上面的理论处理一些图像数据前，我们还需要了解一些几何学知识。给定图像中的一个点，例如第二个视图中的图像点 $\mathbf{x}_2$，利用公式（5.1），我们找到对应第一幅图像的一条直线：

$$\mathbf{x}_2^T\, \boldsymbol{F}\, \mathbf{x}_1 = \boldsymbol{l}_1^T \mathbf{x}_1 = 0$$

其中 $\boldsymbol{l}_1^T \mathbf{x}_1 = 0$ 是第一幅图像中的一条直线。这条线称为对应于点 $\mathbf{x}_2$ 的外极线，也就

注 1：重建的绝对尺度不能恢复，但在实际中，这并不是个问题。

是说，点 $\mathbf{x}_2$ 在第一幅图像中的对应点一定在这条线上。所以，基础矩阵可以将对应点的搜索限制在外极线上。

外极线都经过一点 $\mathbf{e}$，称为外极点。实际上，外极点是另一个照相机光心对应的图像点。外极点可以在我们看到的图像外，这取决于照相机间的相对方向。因为外极点在所有的外极线上，所以 $\boldsymbol{F}\mathbf{e}_1=0$。因此，我们可以通过计算 $\boldsymbol{F}$ 的零向量来计算出外极点。同理，另一个外极点可以通过计算 $\mathbf{e}_2^T\boldsymbol{F}=0$ 得到。外极点和外极线如图 5-1 所示。

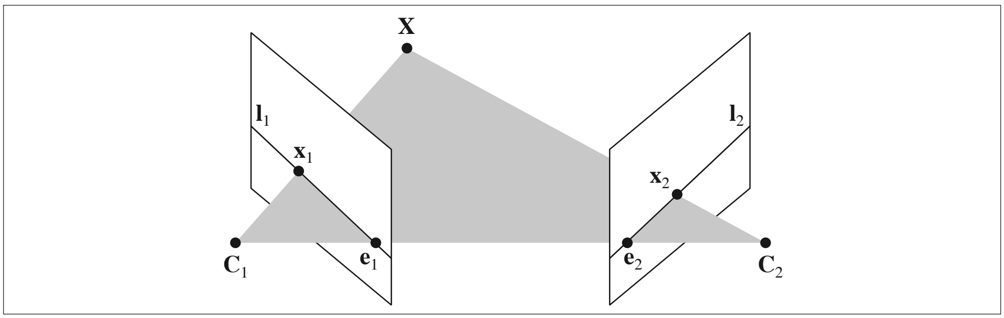


图 5-1：外极几何示意图。在这两个视图中，分别将三维点 X 投影为 x_1 和 x_2。两个照相机中心之间的基线 C_1 和 C_2 与图像平面相交于外极点 e_1 和 e_2。线 l_1 和 l_2 称为外极线

5.1.1 一个简单的数据集

在接下来的几节中，我们需要一个带有图像点、三维点和照相机参数矩阵的数据集。我们这里使用一个牛津多视图数据集；从 http://www.robots.ox.ac.uk/~vgg/data/data-mview.html 可以下载 Merton1 数据的压缩文件。下面的脚本可以加载 Merton1 的数据：

```
import camera

# 载入一些图像
im1 = array(Image.open('images/001.jpg'))
im2 = array(Image.open('images/002.jpg'))

# 载入每个视图的二维点到列表中
points2D = [loadtxt('2D/00'+str(i+1)+'.corners').T for i in range(3)]

# 载入三维点
points3D = loadtxt('3D/p3d').T

# 载入对应
corr = genfromtxt('2D/nview-corners',dtype='int',missing='*')

# 载入照相机矩阵到 Camera 对象列表中
P = [camera.Camera(loadtxt('2D/00'+str(i+1)+'.P')) for i in range(3)]
```

上面的程序会加载前两个图像（共三个）、三个视图中的所有图像特征点[1]、对应不同视图图像点重建后的三维点以及照相机参数矩阵（使用上一章的 Camera 类）。这里我们使用 loadtxt() 函数读取文本文件到 NumPy 数组中。因为并不是所有的点都可见，或都能够成功匹配到所有的视图，所以对应数据里包含了缺失的数据。加载对应数据时需要考虑这一点。genfromtxt() 函数通过将缺失的数值（在文件中用 * 表示）填充为 -1 来解决这个问题。

将上面的代码保存到一个文件，例如 load_vggdata.py，然后使用命令 execfile() 可以很方便地运行上面的脚本，从而获取所有的数据：

```
execfile('load_vggdata.py')
```

下面来可视化这些数据。将三维的点投影到一个视图，然后和观测到的图像点比较：

```
# 将三维点转换成齐次坐标表示，并投影
X = vstack( (points3D,ones(points3D.shape[1])) )
x = P[0].project(X)

# 在视图 1 中绘制点
figure()
imshow(im1)
plot(points2D[0][0],points2D[0][1],'*')
axis('off')

figure()
imshow(im1)
plot(x[0],x[1],'r.')
axis('off')

show()
```

上面的代码绘制出第一个视图以及该视图中的图像点。为比较方便，投影后的点绘制在另一张图上，如图 5-2 所示。如果仔细观察，你会发现第二幅图比第一幅图多一些点。这些多出的点是从视图 2 和视图 3 重建出来的，而不在视图 1 中。

注 1：事实上，这些特征点是 2.1 节中的 Harris 角点。

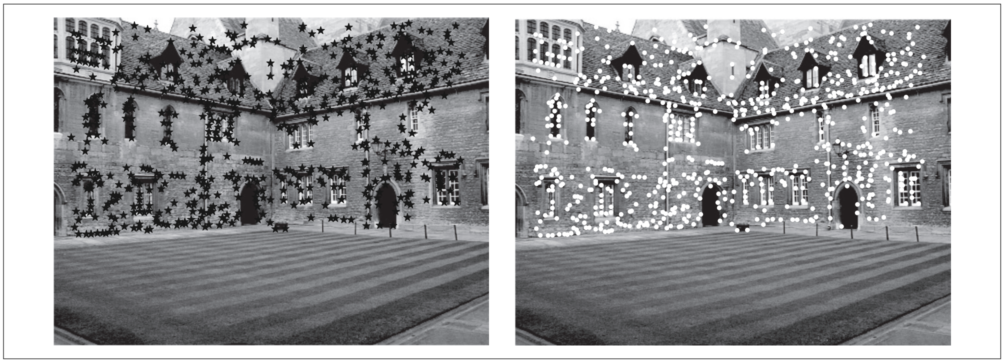


图 5-2：牛津 multi-view 数据集中的 Merton1 数据集：视图 1 与图像点（左）；视图 1 与投影的三维点（右）

5.1.2 用Matplotlib绘制三维数据

为了可视化三维重建结果，我们需要绘制出三维图像。Matplotlib 中的 mplot3d 工具包可以方便地绘制出三维点、线、等轮廓线、表面以及其他基本图形组件，还可以通过图像窗口控件实现三维旋转和缩放。

可以通过在 axes 对象中加上 projection="3d" 关键字实现三维绘图，如下：

```
from mpl_toolkits.mplot3d import axes3d

fig = figure()
ax = fig.gca(projection="3d")

# 生成三维样本点
X,Y,Z = axes3d.get_test_data(0.25)

# 在三维中绘制点
ax.plot(X.flatten(),Y.flatten(),Z.flatten(),'o')

show()
```

get_test_data() 函数在 x, y 空间按照设定的空间间隔参数来产生均匀的采样点。压平这些网格，会产生三列数据点，然后我们可以将其输入 plot() 函数。这样，我们就可以在立体表面上画出三维点。

现在通过画出 Merton 样本数据来观察三维点的效果：

```
# 绘制三维点
from mpl_toolkits.mplot3d import axes3d
```

```
fig = figure()
ax = fig.gca(projection='3d')
ax.plot(points3D[0],points3D[1],points3D[2],'k.')
```

图 5-3 是三个不同视图中的三维图像点。图像窗口和控制界面外观效果像加上三维旋转工具的标准画图窗口。

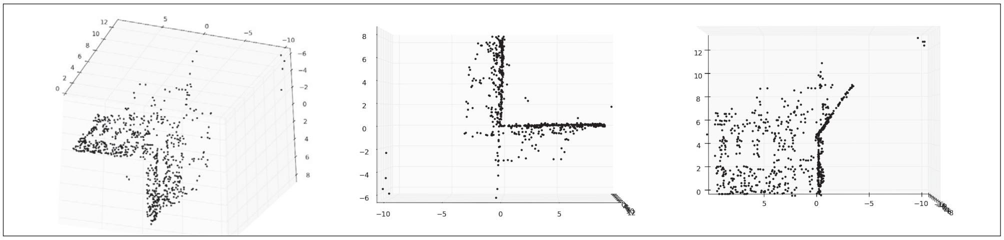


图 5-3：使用 Matplottib 工具包绘制的，牛津 multi-view 数据库中 Mertoln1 数据集的三维点：从上面和侧边观测的视图（左）；俯视的视图，展示了建筑墙体和屋顶上的点（中）；侧视图，展示了一面墙的轮廓，以及另一面墙上点的主视图（右）

5.1.3 计算*F*：八点法

八点法是通过对应点来计算基础矩阵的算法。下面给出概述，更多内容参阅文献 [13] 和文献 [14]。

外极约束（5.1）可以写成线性系统的形式：

$$\begin{bmatrix} x_2^1 x_1^1 & x_2^1 y_1^1 & x_2^1 w_1^1 & \cdots & w_2^1 w_1^1 \\ x_2^2 x_1^2 & x_2^2 y_1^2 & x_2^2 w_1^2 & \cdots & w_2^2 w_1^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_2^n x_1^n & x_2^n y_1^n & x_2^n w_1^n & \cdots & w_2^n w_1^n \end{bmatrix} \begin{bmatrix} \boldsymbol{F}_{11} \\ \boldsymbol{F}_{12} \\ \boldsymbol{F}_{13} \\ \vdots \\ \boldsymbol{F}_{33} \end{bmatrix} = \boldsymbol{A}\boldsymbol{f} = 0$$

其中，$\boldsymbol{f}$ 包含 $\boldsymbol{F}$ 的元素，$\mathbf{x}_1^i = [x_1^i, y_1^i, w_1^i]$ 和 $\mathbf{x}_2^i = [x_2^i, y_2^i, w_2^i]$ 是一对图像点，共有 n 对对应点。基础矩阵中有 9 个元素，由于尺度是任意的，所以只需要 8 个方程。因为算法中需要 8 个对应点来计算基础矩阵 $\boldsymbol{F}$，所以该算法叫做八点法。

新建一个文件 sfm.py，写入下面八点法中最小化 $\|\boldsymbol{A}\boldsymbol{f}\|$ 的函数：

```
def compute_fundamental(x1,x2):
  """ 使用归一化的八点算法，从对应点（x1，x2 3×n 的数组）中计算基础矩阵
    每行由如下构成：
    [x'*x, x'*y' x', y'*x, y'*y, y', x, y, 1]"""

  n = x1.shape[1]
```

```
    if x2.shape[1] != n:
      raise ValueError("Number of points don't match.")

    # 创建方程对应的矩阵
    A = zeros((n,9))
    for i in range(n):
      A[i] = [x1[0,i]*x2[0,i], x1[0,i]*x2[1,i], x1[0,i]*x2[2,i],
          x1[1,i]*x2[0,i], x1[1,i]*x2[1,i], x1[1,i]*x2[2,i],
          x1[2,i]*x2[0,i], x1[2,i]*x2[1,i], x1[2,i]*x2[2,i] ]

    # 计算线性最小二乘解
    U,S,V = linalg.svd(A)
    F = V[-1].reshape(3,3)

    # 受限 F
    # 通过将最后一个奇异值置 0，使秩为 2
    U,S,V = linalg.svd(F)
    S[2] = 0
    F = dot(U,dot(diag(S),V))

    return F
```

我们通常用 SVD 算法来计算最小二乘解。由于上面算法得出的解可能秩不为 2（基础矩阵的秩小于等于 2），所以我们通过将最后一个奇异值置 0 来得到秩最接近 2 的基础矩阵。这是个很有用的技巧。上面的函数忽略了一个重要的步骤：对图像坐标进行归一化，这可能会带来数值问题。我们会在后面加以解决。

5.1.4 外极点和外极线

本节开始提到过，外极点满足 $\boldsymbol{F}\mathbf{e}_1=0$，因此可以通过计算 $\boldsymbol{F}$ 的零空间来得到。把下面的函数添加到 sfm.py 中：

```
def compute_epipole(F):
  """ 从基础矩阵 F 中计算右极点（可以使用 F.T 获得左极点）"""

  # 返回 F 的零空间（Fx=0）
  U,S,V = linalg.svd(F)
  e = V[-1]
return e/e[2]
```

如果想获得另一幅图像的外极点（对应左零空间的外极点），只需将 $\boldsymbol{F}$ 转置后输入上述函数即可。

我们可以在之前样本数据集的前两个视图上运行这两个函数：

```
import sfm

# 在前两个视图中点的索引
ndx = (corr[:,0]>=0) & (corr[:,1]>=0)

# 获得坐标，并将其用齐次坐标表示
x1 = points2D[0][:,corr[ndx,0]]
x1 = vstack( (x1,ones(x1.shape[1])) )
x2 = points2D[1][:,corr[ndx,1]]
x2 = vstack( (x2,ones(x2.shape[1])) )

# 计算 F
F = sfm.compute_fundamental(x1,x2)

# 计算极点
e = sfm.compute_epipole(F)

# 绘制图像
figure()
imshow(im1)

# 分别绘制每条线，这样会绘制出很漂亮的颜色
for i in range(5):
  sfm.plot_epipolar_line(im1,F,x2[:,i],e,False)
axis('off')

figure()
imshow(im2)
# 分别绘制每个点，这样会绘制出和线同样的颜色
for i in range(5):
  plot(x2[0,i],x2[1,i],'o')
axis('off')

show()
```

首先，选择两幅图像的对应点，然后将它们转换为齐次坐标。这里的对应点是从一个文本文件中读取得到的；而实际上，我们可以按照第 2 章的方法提取图像特征，然后通过匹配来找到它们。由于缺失的数据在对应列表 corr 中为 −1，所以程序中有可能选取这些点。因此，上面的程序通过数组操作符 & 只选取了索引大于等于 0 的点。

最后，我们在第一个视图中画出了前 5 个外极线，在第二个视图中画出了对应匹配点。这里我们主要借助 plot() 函数：

```
def plot_epipolar_line(im,F,x,epipole=None,show_epipole=True):
  """ 在图像中，绘制外极点和外极线 F×x=0。F 是基础矩阵，x 是另一幅图像中的点 """
```

```
m,n = im.shape[:2]
line = dot(F,x)

# 外极线参数和值
t = linspace(0,n,100)
lt = array([(line[2]+line[0]*tt)/(-line[1]) for tt in t])

# 仅仅处理位于图像内部的点和线
ndx = (lt>=0) & (lt<m)
plot(t[ndx],lt[ndx],linewidth=2)

if show_epipole:
  if epipole is None:
    epipole = compute_epipole(F)
  plot(epipole[0]/epipole[2],epipole[1]/epipole[2],'r*')
```

上面的函数将 x 轴的范围作为直线的参数，因此直线超出图像边界的部分会被截断。如果 show_epipole 为真，外极点也会被画出来（如果输入参数中没有外极点，外极点会在程序中计算获得）。程序结果如图 5-4 所示。在两幅图中，用不同的颜色将点和相应的外极线对应起来。

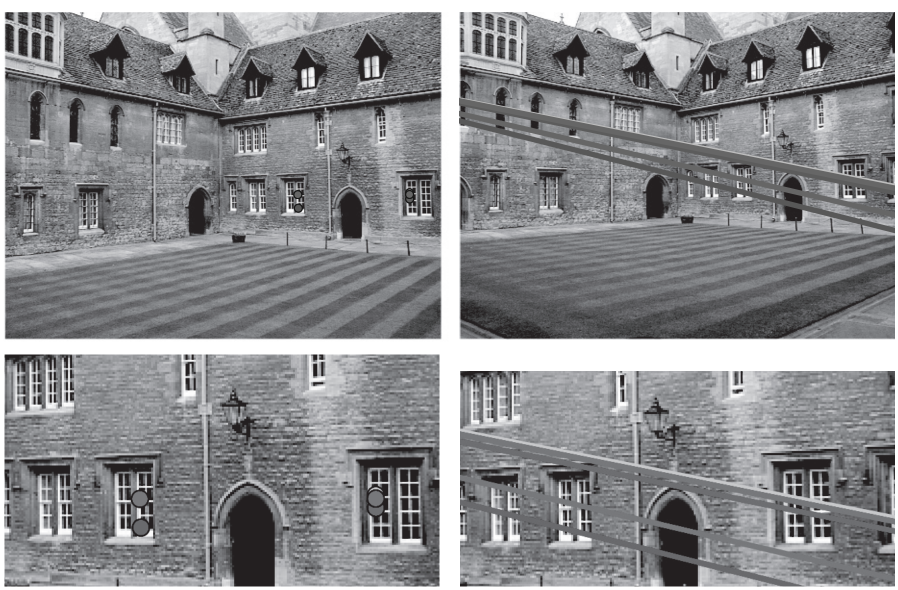


图 5-4：视图 1 中的外极线示为 Merton1 数据视图 2 中 5 个点对应的外极线。下面一行为这些点附近的特写。可以看到，这些线在图像外左侧位置将相交于一点。这些线表明点之间的对应可以在另外一幅图像中找到（线和点之间的颜色编码匹配）

5.2 照相机和三维结构的计算

上一节讲述了视图和基础矩阵、外极线计算方法的关系。这一节我们将简单地介绍计算照相机参数和三维结构的工具。

5.2.1 三角剖分

给定照相机参数模型，图像点可以通过三角剖分来恢复出这些点的三维位置。基本的算法思想如下。

对于两个照相机 P_1 和 P_2 的视图，三维实物点 $\mathbf{X}$ 的投影点为 $\mathbf{x}_1$ 和 $\mathbf{x}_2$（这里用齐次坐标表示），照相机方程（4.1）定义了下列关系：

$$\begin{bmatrix} P_1 & -\mathbf{x}_1 & 0 \\ P_2 & 0 & -\mathbf{x}_2 \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ \lambda_1 \\ \lambda_2 \end{bmatrix} = 0$$

由于图像噪声、照相机参数误差和其他系统误差，上面的方程可能没有精确解。我们可以通过 SVD 算法来得到三维点的最小二乘估值。

下面的函数用于计算一个点对的最小二乘三角剖分，把它添加到 sfm.py 中：

```
def triangulate_point(x1,x2,P1,P2):
  """ 使用最小二乘解，绘制点对的三角剖分 """

  M = zeros((6,6))
  M[:3,:4] = P1
  M[3:,:4] = P2
  M[:3,4] = -x1
  M[3:,5] = -x2

  U,S,V = linalg.svd(M)
  X = V[-1,:4]

  return X / X[3]
```

最后一个特征向量的前 4 个值就是齐次坐标系下的对应三维坐标。我们可以增加下面的函数来实现多个点的三角剖分：

```
def triangulate(x1,x2,P1,P2):
  """ x1 和 x2（3×n 的齐次坐标表示）中点的二视图三角剖分 """

  n = x1.shape[1]
  if x2.shape[1] != n:
```

```
        raise ValueError("Number of points don't match.")

    X = [ triangulate_point(x1[:,i],x2[:,i],P1,P2) for i in range(n)]
    return array(X).T
```

这个函数的输入是两个图像点数组，输出为一个三维坐标数组。

我们可以利用下面的代码来实现 Merton1 数据集上的三角剖分：

```
import sfm
# 前两个视图中点的索引
ndx = (corr[:,0]>=0) & (corr[:,1]>=0)

# 获取坐标，并用齐次坐标表示
x1 = points2D[0][:,corr[ndx,0]]
x1 = vstack( (x1,ones(x1.shape[1])) )
x2 = points2D[1][:,corr[ndx,1]]
x2 = vstack( (x2,ones(x2.shape[1])) )

Xtrue = points3D[:,ndx]
Xtrue = vstack( (Xtrue,ones(Xtrue.shape[1])) )

# 检查前三个点
Xest = sfm.triangulate(x1,x2,P[0].P,P[1].P)
print Xest[:,:3]
print Xtrue[:,:3]

# 绘制图像
from mpl_toolkits.mplot3d import axes3d
fig = figure()
ax = fig.gca(projection='3d')
ax.plot(Xest[0],Xest[1],Xest[2],'ko')
ax.plot(Xtrue[0],Xtrue[1],Xtrue[2],'r.')
axis('equal')

show()
```

上面的代码首先利用前两个视图的信息来对图像点进行三角剖分，然后把前三个图像点的齐次坐标输出到控制台，最后绘制出恢复的最接近三维图像点。输出到控制台的信息如下：

```
[[ 1.03743725  1.56125273  1.40720017]
 [-0.57574987 -0.55504127 -0.46523952]
 [ 3.44173797  3.44249282  7.53176488]
 [ 1.          1.          1.        ]]
[[ 1.0378863   1.5606923   1.4071907 ]
```

```
[-0.54627892 -0.5211711  -0.46371818]
[ 3.4601538   3.4636809   7.5323397 ]
[ 1.          1.          1.        ]]
```

算法估计出的三维图像点和实际图像点很接近。如图 5-5 所示，估计点和实际点可以很好地匹配。

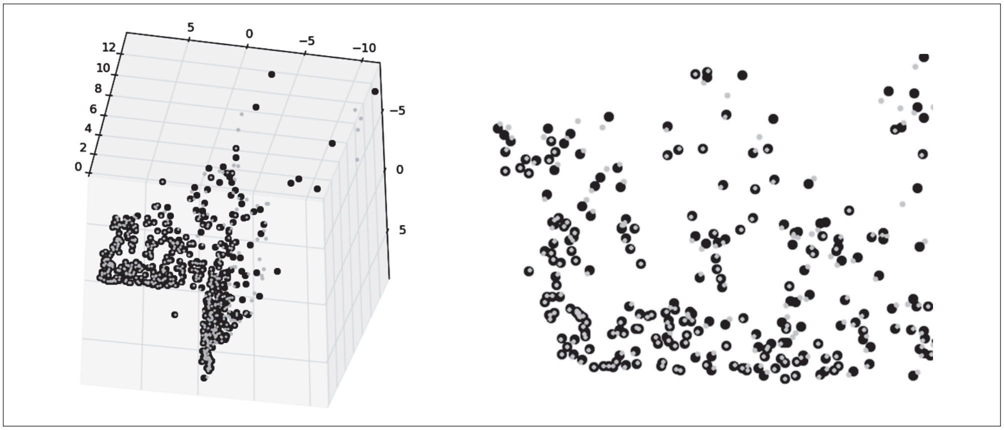


图 5-5：使用照相机矩阵和点的对应关系来三角剖分这些数据点。黑色的圆圈表示估计的点，灰色的点表示真实点。上方一侧的视图（左）。一个坐标平面观看这些数据点的近景图像（右）

5.2.2 由三维点计算照相机矩阵

如果已经知道了一些三维点及其图像投影，我们可以使用直接线性变换的方法来计算照相机矩阵 $\boldsymbol{P}$。本质上，这是三角剖分方法的逆问题，有时我们将其称为*照相机反切法*。利用该方法恢复照相机矩阵同样也是一个最小二乘问题。

我们从照相机方程（4.1）可以得出，每个三维点 $\mathbf{X}_i$（齐次坐标系下）按照 $\lambda_i\mathbf{x}_i=\boldsymbol{P}\mathbf{X}_i$ 投影到图像点 $\mathbf{x}_i=[x_i, y_i, 1]$，相应的点满足下面的关系：

$$\begin{bmatrix} \mathbf{X}_1^T & 0 & 0 & -x_1 & 0 & 0 & \cdots \\ 0 & \mathbf{X}_1^T & 0 & -y_1 & 0 & 0 & \cdots \\ 0 & 0 & \mathbf{X}_1^T & -1 & 0 & 0 & \cdots \\ \mathbf{X}_2^T & 0 & 0 & 0 & -x_2 & 0 & \cdots \\ 0 & \mathbf{X}_2^T & 0 & 0 & -y_2 & 0 & \cdots \\ 0 & 0 & \mathbf{X}_2^T & 0 & -1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \mathbf{p}_3^T \\ \lambda_1 \\ \lambda_2 \\ \vdots \end{bmatrix} = 0$$

其中 $\mathbf{p}_1$、$\mathbf{p}_2$ 和 $\mathbf{p}_3$ 是矩阵 $\boldsymbol{P}$ 的三行。上面的式子可以写得更紧凑，如下所示：

$$\boldsymbol{M}\mathbf{v}=0$$

然后，我们可以使用 SVD 分解估计出照相机矩阵。利用上面讲述的矩阵操作，我们可以直接写出相应的代码。将下面的函数添加到 sfm.py 文件中：

```
def compute_P(x,X):
  """ 由二维 - 三维对应对（齐次坐标表示）计算照相机矩阵 """

  n = x.shape[1]
  if X.shape[1] != n:
    raise ValueError("Number of points don't match.")

  # 创建用于计算 DLT 解的矩阵
  M = zeros((3*n,12+n))
  for i in range(n):
    M[3*i,0:4] = X[:,i]
    M[3*i+1,4:8] = X[:,i]
    M[3*i+2,8:12] = X[:,i]
    M[3*i:3*i+3,i+12] = -x[:,i]

  U,S,V = linalg.svd(M)

  return V[-1,:12].reshape((3,4))
```

该函数的输入参数为图像点和三维点，构造出上述所示的 $\boldsymbol{M}$ 矩阵。最后一个特征向量的前 12 个元素是照相机矩阵的元素，经过重新排列成矩阵形状后返回。

下面，在我们的样本数据集上测试算法的性能。下面的代码会选出第一个视图中的一些可见点（使用对应列表中缺失的数值），将它们转换为齐次坐标表示，然后估计照相机矩阵：

```
import sfm, camera

corr = corr[:,0] # 视图 1
ndx3D = where(corr>=0)[0] # 丢失的数值为 -1
ndx2D = corr[ndx3D]

# 选取可见点，并用齐次坐标表示
x = points2D[0][:,ndx2D] # 视图 1
x = vstack( (x,ones(x.shape[1])) )
X = points3D[:,ndx3D]
X = vstack( (X,ones(X.shape[1])) )

# 估计 P
Pest = camera.Camera(sfm.compute_P(x,X))

# 比较!
print Pest.P / Pest.P[2,3]
```

```
print P[0].P / P[0].P[2,3]

xest = Pest.project(X)

# 绘制图像
figure()
imshow(im1)
plot(x[0],x[1],'bo')
plot(xest[0],xest[1],'r.')
axis('off')

show()
```

为了检查照相机矩阵的正确性，将它们以归一化的格式（除以最后一个元素）打印到控制台。输出如下所示：

```
[[  1.06520794e+00 -5.23431275e+01  2.06902749e+01 5.08729305e+02]
 [ -5.05773115e+01 -1.33243276e+01 -1.47388537e+01 4.79178838e+02]
 [  3.05121915e-03 -3.19264684e-02 -3.43703738e-02 1.00000000e+00]]
[[  1.06774679e+00 -5.23448212e+01  2.06926980e+01 5.08764487e+02]
 [ -5.05834364e+01 -1.33201976e+01 -1.47406641e+01 4.79228998e+02]
 [  3.06792659e-03 -3.19008054e-02 -3.43665129e-02 1.00000000e+00]]
```

上面是估计出的照相机矩阵，下面是数据集的创建者计算出的照相机矩阵。可以看到，它们的元素几乎完全相同。最后，使用估计出的照相机矩阵投影这些三维点，然后绘制出投影后的结果。结果如图 5-6 所示，真实点用圆圈表示，估计出的照相机投影点用点表示。

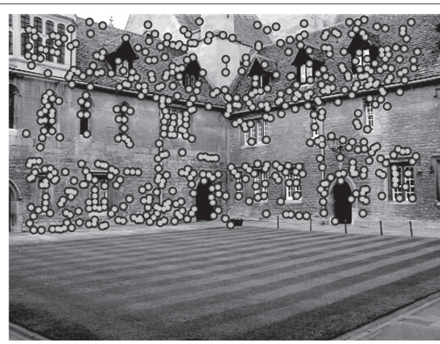


图 5-6：使用估计出的照相机矩阵来计算视图 1 中的投影点

5.2.3　由基础矩阵计算照相机矩阵

在两个视图的场景中，照相机矩阵可以由基础矩阵恢复出来。假设第一个照相机矩

阵归一化为 P_1=[I|0]，现在我们需要计算出第二个照相机矩阵 P_2。研究分为两类，未标定的情况和已标定的情况。

1. 未标定的情况——投影重建

在没有任何照相机内参数知识的情况下，照相机矩阵只能通过射影变换恢复出来。也就是说，如果利用照相机的信息来重建三维点，那么该重建只能由射影变换计算出来（你可以得到整个投影场景中无畸变的重建点）。在这里，我们不考虑角度和距离。

因此，在无标定的情况下，第二个照相机矩阵可以使用一个（3×3）的射影变换得出。一个简单的方法是：

$$\boldsymbol{P}_2=[\boldsymbol{S}_{\mathbf{e}}\boldsymbol{F}|\mathbf{e}]$$

其中，$\mathbf{e}$ 是左极点，满足 $\mathbf{e}^T\boldsymbol{F}=0$，$\boldsymbol{S}_{\mathbf{e}}$ 是如公式（5.2）所示的反对称矩阵。请注意，使用该矩阵做出的三角形剖分很有可能会发生畸变，如倾斜的重建。

下面是具体的代码：

```
def compute_P_from_fundamental(F):
  """ 从基础矩阵中计算第二个照相机矩阵（假设 P1 = [I 0]）"""

  e = compute_epipole(F.T) # 左极点
  Te = skew(e)
  return vstack((dot(Te,F.T).T,e)).T
```

这里，我们使用的 skew() 函数定义如下：

```
def skew(a):
  """ 反对称矩阵 A，使得对于每个 v 有 a×v=Av """

  return array([[0,-a[2],a[1]],[a[2],0,-a[0]],[-a[1],a[0],0]])
```

将上面的两个函数添加到 sfm.py 文件中。

2. 已标定的情况——度量重建

在已经标定的情况下，重建会保持欧式空间中的一些度量特性（除了全局的尺度参数）。在重建三维场景中，已标定的例子更为有趣。

给定标定矩阵 $\boldsymbol{K}$，我们可以将它的逆 $\boldsymbol{K}^{-1}$ 作用于图像点 $\mathbf{x}_K=\boldsymbol{K}^{-1}\mathbf{x}$，因此，在新的图像坐标系下，照相机方程变为：

$$\mathbf{x}_K = \boldsymbol{K}^{-1}\boldsymbol{K}[\boldsymbol{R}|\mathbf{t}]\mathbf{X} = [\boldsymbol{R}|\mathbf{t}]\mathbf{X}$$

在新的图像坐标系下，点同样满足之前的基础矩阵方程：

$$\mathbf{x}_{K_2}^T \boldsymbol{F} \mathbf{x}_{K_1} = 0$$

在标定归一化的坐标系下，基础矩阵称为本质矩阵。为了区别为标定后的情况，以及归一化了的图像坐标，我们通常将其记为 E，而非 F。

从本质矩阵恢复出的照相机矩阵中存在度量关系，但有四个可能解。因为只有一个解产生位于两个照相机前的场景，所以我们可以轻松地从中选出来。

下面是计算这 4 个解的算法（参阅文献 [13] 获取更多细节）。将该函数添加到 sfm.py 文件中：

```
def compute_P_from_essential(E):
  """ 从本质矩阵中计算第二个照相机矩阵（假设 P1 = [I 0]）
    输出为 4 个可能的照相机矩阵列表 """

  # 保证 E 的秩为 2
  U,S,V = svd(E)
  if det(dot(U,V))<0:
    V = -V
  E = dot(U,dot(diag([1,1,0]),V))

  # 创建矩阵（Hartley）
  Z = skew([0,0,-1])
  W = array([[0,-1,0],[1,0,0],[0,0,1]])

  # 返回所有（4 个）解
  P2 = [vstack((dot(U,dot(W,V)).T,U[:,2])).T,
       vstack((dot(U,dot(W,V)).T,-U[:,2])).T,
       vstack((dot(U,dot(W.T,V)).T,U[:,2])).T,
       vstack((dot(U,dot(W.T,V)).T,-U[:,2])).T]

  return P2
```

首先，该函数确保本质矩阵的秩为 2（本质矩阵有两个相等的非零奇异值）。然后，按照文献 [13] 中的方法，计算出这 4 个解。关于如何挑选正确的解，我们将在后面的例子中讲解。

本节涵盖了从图像集计算出三维重建所需的所有理论。

5.3　多视图重建

下面让我们来看，如何使用上面的理论从多幅图像中计算出真实的三维重建。由于

照相机的运动给我们提供了三维结构，所以这样计算三维重建的方法通常称为 SfM（Structure from Motion，从运动中恢复结构）。

假设照相机已经标定，计算重建可以分为下面 4 个步骤：

(1) 检测特征点，然后在两幅图像间匹配；
(2) 由匹配计算基础矩阵；
(3) 由基础矩阵计算照相机矩阵；
(4) 三角剖分这些三维点。

我们已经具备了完成上面 4 个步骤所需的所有工具。但是当图像间的点对应包含不正确的匹配时，我们需要一个稳健的方法来计算基础矩阵。

5.3.1 稳健估计基础矩阵

类似于稳健计算单应性矩阵（3.3 节），当存在噪声和不正确的匹配时，我们需要估计基础矩阵。和前面的方法一样，我们使用 RANSAC 方法，只不过这次结合了八点算法。注意，八点算法在平面场景中会失效，所以，如果场景点都位于平面上，就不能使用该算法。

将下面的类添加到 sfm.py 文件中：

```
class RansacModel(object):
  """ 用从 http://www.scipy.org/Cookbook/RANSAC 下载的 ransac.py 计算基础矩阵的类 """

  def __init__(self,debug=False):
    self.debug = debug

  def fit(self,data):
    """ 使用选择的 8 个对应计算基础矩阵 """

    # 转置，并将数据分成两个点集
    data = data.T
    x1 = data[:3,:8]
    x2 = data[3:,:8]

    # 估计基础矩阵，并返回
    F = compute_fundamental_normalized(x1,x2)
    return F

  def get_error(self,data,F):
    """ 计算所有对应的 x^T F x，并返回每个变换后点的误差 """

    # 转置，并将数据分成两个点集
```

```
    data = data.T
    x1 = data[:3]
    x2 = data[3:]

    # 将 Sampson 距离用作误差度量
    Fx1 = dot(F,x1)
    Fx2 = dot(F,x2)
    denom = Fx1[0]**2 + Fx1[1]**2 + Fx2[0]**2 + Fx2[1]**2
    err = ( diag(dot(x1.T,dot(F,x2))) )**2 / denom

    # 返回每个点的误差
    return err
```

和之前一样，我们需要 `fit()` 和 `get_error()` 方法。这里采用的错误衡量方法是 Sampson 距离（参阅文献 [13]）。`fit()` 方法会选择 8 个点，然后使用归一化的八点算法：

```
def compute_fundamental_normalized(x1,x2):
  """ 使用归一化的八点算法，由对应点（x1，x2 3×n 的数组）计算基础矩阵 """

  n = x1.shape[1]
  if x2.shape[1] != n:
  raise ValueError("Number of points don't match.")

  # 归一化图像坐标
  x1 = x1 / x1[2]
  mean_1 = mean(x1[:2],axis=1)
  S1 = sqrt(2) / std(x1[:2])
  T1 = array([[S1,0,-S1*mean_1[0]],[0,S1,-S1*mean_1[1]],[0,0,1]])
  x1 = dot(T1,x1)

  x2 = x2 / x2[2]
  mean_2 = mean(x2[:2],axis=1)
  S2 = sqrt(2) / std(x2[:2])
  T2 = array([[S2,0,-S2*mean_2[0]],[0,S2,-S2*mean_2[1]],[0,0,1]])
  x2 = dot(T2,x2)

  # 使用归一化的坐标计算 F
  F = compute_fundamental(x1,x2)

  # 反归一化
  F = dot(T1.T,dot(F,T2))

  return F/F[2,2]
```

该函数将图像点归一化为零均值固定方差。

接下来，我们在函数中使用该类。将下面的函数添加到 sfm.py 文件中：

```
def F_from_ransac(x1,x2,model,maxiter=5000,match_theshold=1e-6):
  """ 使用 RANSAN 方法（ransac.py，来自 http://www.scipy.org/Cookbook/RANSAC），
    从点对应中稳健地估计基础矩阵 F
    输入：使用齐次坐标表示的点 x1，x2（3×n 的数组）"""

    import ransac

    data = vstack((x1,x2))

    # 计算 F，并返回正确点索引
    F,ransac_data = ransac.ransac(data.T,model,8,maxiter,match_theshold,20,
                                  return_all=True)
    return F, ransac_data['inliers']
```

这里，我们返回最佳基础矩阵 $\boldsymbol{F}$，以及正确点的索引，这样可以知道哪些匹配和 $\boldsymbol{F}$ 矩阵是一致的。与单应性矩阵估计相比，我们增加了默认的最大迭代次数，改变了匹配的阈值（之前使用像素，现在使用 Sampson 距离来衡量）。

5.3.2　三维重建示例

在本节中，我们将观察一个重建三维场景的完整例子。我们使用由已知标定矩阵的照相机拍摄的两幅图像。该图像是著名的恶魔岛监狱，如图 5-7 所示。[1]

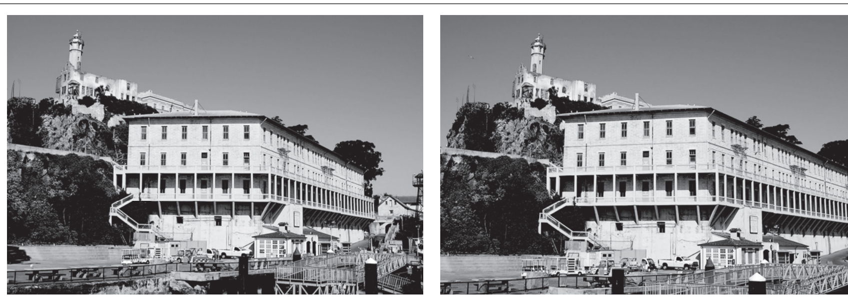


图 5-7：在一个场景中使用不同视角拍摄图像对

我们将该处理代码分成若干块，使得代码更容易理解。首先，提取、匹配特征，然后估计基础矩阵和照相机矩阵：

```
import homography
import sfm
import sift
```

注 1：图像由 Carl Olsson 提供（参见 http://www.maths.lth.se/matematiklth/personal/calle/）。

```
# 标定矩阵
K = array([[2394,0,932],[0,2398,628],[0,0,1]])

# 载入图像，并计算特征
im1 = array(Image.open('alcatraz1.jpg'))
sift.process_image('alcatraz1.jpg','im1.sift')
l1,d1 = sift.read_features_from_file('im1.sift')

im2 = array(Image.open('alcatraz2.jpg'))
sift.process_image('alcatraz2.jpg','im2.sift')
l2,d2 = sift.read_features_from_file('im2.sift')

# 匹配特征
matches = sift.match_twosided(d1,d2)
ndx = matches.nonzero()[0]

# 使用齐次坐标表示，并使用 inv(K) 归一化
x1 = homography.make_homog(l1[ndx,:2].T)
ndx2 = [int(matches[i]) for i in ndx]
x2 = homography.make_homog(l2[ndx2,:2].T)

x1n = dot(inv(K),x1)
x2n = dot(inv(K),x2)

# 使用 RANSAC 方法估计 E
model = sfm.RansacModel()
E,inliers = sfm.F_from_ransac(x1n,x2n,model)

# 计算照相机矩阵（P2 是 4 个解的列表）
P1 = array([[1,0,0,0],[0,1,0,0],[0,0,1,0]])
P2 = sfm.compute_P_from_essential(E)
```

现在，我们已经获得了标定矩阵，所以只对矩阵 $\boldsymbol{K}$ 进行硬编码。与之前的例子一样，我们挑选属于匹配关系的特定点。然后使用 $\boldsymbol{K}^{-1}$ 来对其进行归一化，并使用归一化的八点算法来运行 RANSAC 估计。因为该点已经归一化，所以这里会返回一个本质矩阵。因为我们将会使用到的正确的匹配点，所以需要保存正确匹配点的索引。从本质矩阵出发，我们可以计算出第二个照相机矩阵的四个可能解。

从照相机矩阵的列表中，挑选出经过三角剖分后，在两个照相机前均含有最多场景点的照相机矩阵：

```
# 选取点在照相机前的解
ind = 0
maxres = 0
for i in range(4):
```

```
    # 三角剖分正确点，并计算每个照相机的深度
    X = sfm.triangulate(x1n[:,inliers],x2n[:,inliers],P1,P2[i])
    d1 = dot(P1,X)[2]
    d2 = dot(P2[i],X)[2]
    if sum(d1>0)+sum(d2>0) > maxres:
      maxres = sum(d1>0)+sum(d2>0)
      ind = i
      infront = (d1>0) & (d2>0)

    # 三角剖分正确点，并移除不在所有照相机前面的点
    X = sfm.triangulate(x1n[:,inliers],x2n[:,inliers],P1,P2[ind])
    X = X[:,infront]
```

循环遍历这 4 个解，每次对对应于正确点的三维点进行三角剖分。将三角剖分后的图像投影回图像后，深度的符号由每个图像点的第三个数值给出。我们保存了正向最大深度的索引；对于和最优解一致的点，用相应的布尔量保存了信息，这样可以取出真正在照相机前面的点。因为所有估计中都存在噪声和误差，所以即便使用正确的照相机矩阵，也存在一些点仍位于某个照相机后面的风险。首先获得正确的解，然后对这些正确的点进行三角剖分，最后保留位于照相机前方的点。

现在可以绘制出该三维重建：

```
# 绘制三维图像
from mpl_toolkits.mplot3d import axes3d

fig = figure()
ax = fig.gca(projection='3d')
ax.plot(-X[0],X[1],X[2],'k.')
axis('off')
```

和我们的坐标系相比，使用 mplot3d 绘制三维图像需要将第一个坐标值取相反数，所以这里改变其符号。

然后，可以在每个视图中绘制出二次投影：

```
# 绘制 X 的投影
import camera

# 绘制三维点
cam1 = camera.Camera(P1)
cam2 = camera.Camera(P2[ind])
x1p = cam1.project(X)
x2p = cam2.project(X)

# 反 K 归一化
```

```
x1p = dot(K,x1p)
x2p = dot(K,x2p)

figure()
imshow(im1)
gray()
plot(x1p[0],x1p[1],'o')
plot(x1[0],x1[1],'r.')
axis('off')

figure()
imshow(im2)
gray()
plot(x2p[0],x2p[1],'o')
plot(x2[0],x2[1],'r.')
axis('off')
show()
```

将这些三维点投影后，可以通过乘以标定矩阵的方式来弥补初始归一化对点的影响。

结果输出如图 5-8 所示。可以看到，二次投影后的点和原始特征位置不完全匹配，但是相当接近。当然，可以进一步调整照相机矩阵来提高重建和二次投影的性能，但是这超出了这个简单例子的范畴。

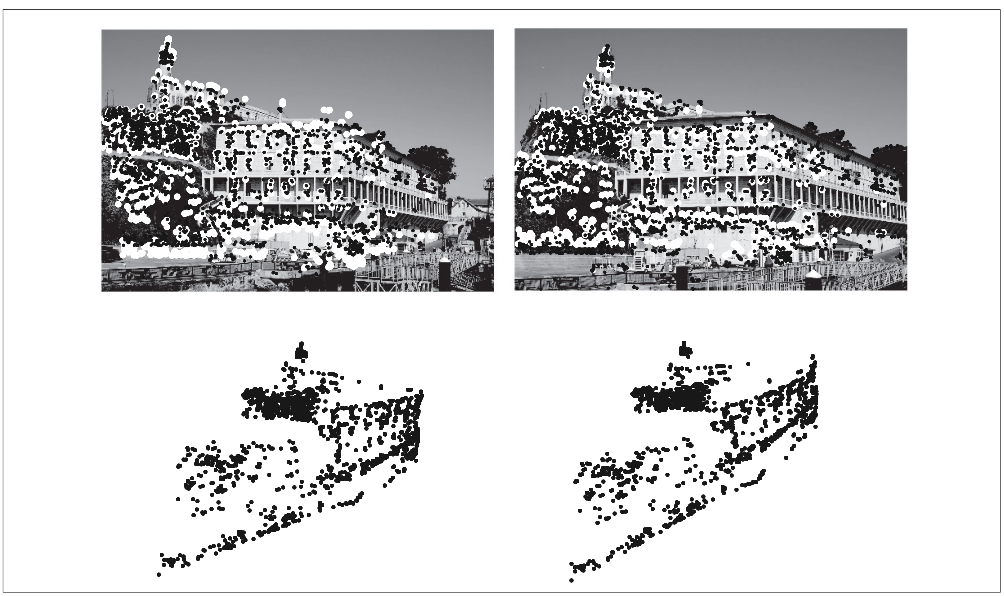


图 5-8：使用图像匹配从一对图像中计算三维重建：带有特征点（黑色）和二次投影重建的三维点（白色）的两幅图像（上）；三维重建（下）

5.3.3 多视图的扩展示例

多视图重建有一些步骤和进一步的扩展，但是不能在本书中全部介绍。为了方便读者进一步阅读，下面提供关于一些有关内容及其参考文献。

1. 多视图

当我们有同一场景的多个视图时，三维重建会变得更准确，包括更多的细节信息。因为基础矩阵只和一对视图相关，所以该过程带有很多图像，和之前的处理有些不同。

对于视频序列，我们可以使用时序关系，在连续的帧对中匹配特征。相对方位需要从每个帧对增量地加入下一个帧对，（与图 3-12 中我们在全景图例子中加入单应性矩阵相同）。该方法非常有效，同时可以使用跟踪有效地找到对应点（关于跟踪的更多知识，参阅 10.4 节）。一个问题是误差会在加入的视图中积累。该问题可以由最后的优化步骤来解决，参见下文。

对于静止的图像，一个办法是找到一个中央参考视图，然后计算与之有关的所有其他照相机矩阵。另一个办法是计算一个图像对的照相机矩阵和三维重建，然后增量地加入新的图像和三维点，参见参考文献 [34]。另外，还有一些方法可以同时由三个视图计算三维重建和照相机位置（参阅参考文献 [13]）；但除此之外，我们还需要使用增量的方法。

2. 光束法平差

从图 5-8 简单三维重建的例子，我们可以清楚地看出，恢复出的点的位置和由估计的基础矩阵计算出的照相机矩阵都存在误差。在多个视图的计算中，这些误差会进一步累积。因此，多视图重建的最后一步，通常是通过优化三维点的位置和照相机参数来减少二次投影误差。该过程称为*光束法平差*。更多内容参阅参考文献 [13] 和 [35]，或者可以从 http://en.wikipedia.org/wiki/Bundle_adjustment 参阅该方法的概述。

3. 自标定

在未标定照相机的情形中，有时可以从图像特征来计算标定矩阵。该过程称为*自标定*。根据在照相机标定矩阵参数上做出的假设，以及可用的图像数据的类型（特征匹配、平行线、平面等），我们有很多不同的自标定算法。感兴趣的读者可以参阅参考文献 [13] 及参考文献 [26] 第 6 章的内容。

作为标定的附注，值得一提的是一个叫做 extract_focal.pl 的有用脚本，它是 SfM 系统（http://phototour.cs.washington.edu/bundler/）的一部分。对于常见的照相机，该脚本基于图像 EXIF 数据，使用一个查找表来估计焦距。

5.4 立体图像

一个多视图成像的特殊例子是*立体视觉*（或者立体成像），即使用两台只有水平（向一侧）偏移的照相机观测同一场景。当照相机的位置如上设置，两幅图像具有相同的图像平面，图像的行是垂直对齐的，那么称图像对是经过矫正的。该设置在机器人学中很常见，常被称为*立体平台*。

通过将图像扭曲到公共的平面上，使外极线位于图像行上，任何立体照相机设置都能得到矫正（我们通常构建立体平台来产生经过矫正的图像对）。这超出了本章节的范围，感兴趣的读者可以参阅参考文献 [13] 第 303 页，或者参考文献 [3] 第 430 页。

假设两幅图像经过了矫正，那么对应点的寻找限制在图像的同一行上。一旦找到对应点，由于深度是和偏移成正比的，那么深度（Z 坐标）可以直接由水平偏移来计算，

$$Z = \frac{fb}{x_l - x_r}$$

其中，f 是经过矫正图像的焦距，b 是两个照相机中心之间的距离，x_1 和 x_r 是左右两幅图像中对应点的 x 坐标。分开照相机中心的距离称为*基线*。矫正后的立体照相机设置如图 5-9 所示。

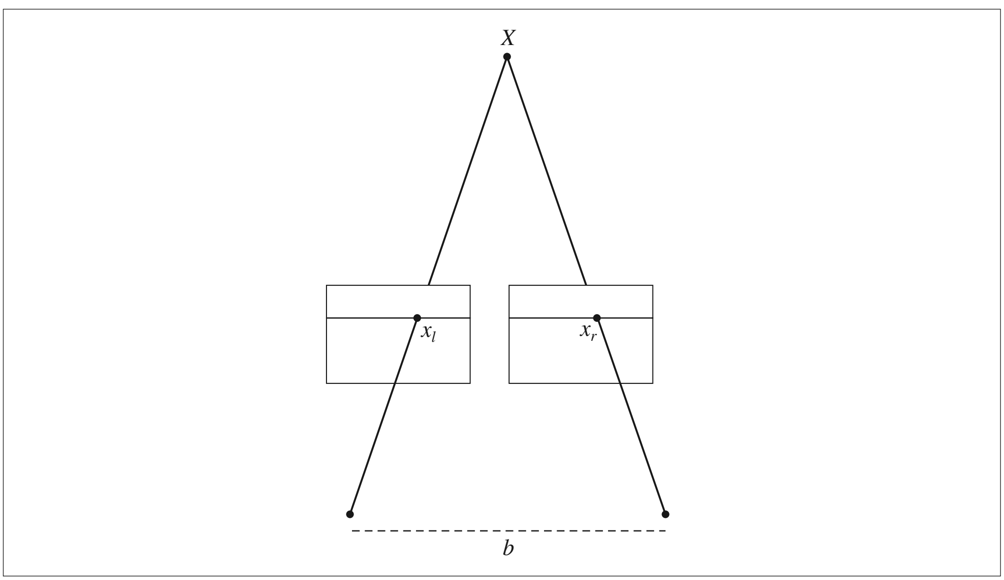


图 5-9：矫正后立体照相机设置的示意图，其中对应点位于两幅图像的同一行

立体重建（有时称为致密深度重建）就是恢复深度图（或者相反，视差图），图像中每个像素的深度（或者视差）都需要计算出来。这是计算机视觉中的经典问题，有很多算法可以解决该问题。米德尔伯里学院立体视觉网页（http://vision.middlebury.edu/stereo/）保持更新最佳算法的评估，以及该算法的代码和许多细节描述。在下一节中，我们会讲解基于归一化互相关的立体重建算法。

计算视差图

在该立体重建算法中，我们将对于每个像素尝试不同的偏移，并按照局部图像周围归一化的互相关值，选择具有最好分数的偏移，然后记录下该最佳偏移。因为每个偏移在某种程度上对应于一个平面，所以该过程有时称为扫平面法。虽然该方法并不是立体重建中最好的方法，但是非常简单，通常会得出令人满意的结果。

当密集地应用在图像中时，归一化的互相关值可以很快地计算出来。这和我们在第 2 章中应用于稀疏点对应的不同。我们使用每个像素周围的图像块（根本上说，是局部周边图像）来计算归一化的互相关。对于这里的情形，我们可以在像素周围重新写出公式（2.3）中的 NCC，如下所示

$$\mathrm{ncc}(\boldsymbol{I}_1,\boldsymbol{I}_2)=\frac{\sum_{\mathbf{x}}(\boldsymbol{I}_1(\mathbf{x})-\mu_1)(\boldsymbol{I}_2(\mathbf{x})-\mu_2)}{\sqrt{\sum_{\mathbf{x}}(\boldsymbol{I}_1(\mathbf{x})-\mu_1)^2\sum_{\mathbf{x}}(\boldsymbol{I}_2(\mathbf{x})-\mu_2)^2}}$$

我们在前面跳过了归一化常数（这里不需要），然后对该像素周围局部像素块中的像素求和。

现在，我们要将图像中的每个像素进行该操作。这三个求和操作是在局部图像块区域上进行的，我们可以使用图像滤波器来快速计算，这与我们在模糊和求导操作中使用的技巧一样。`ndimage.filters` 模块中的 `uniform_filter()` 函数可以在一个矩形图像块中计算相加。

下面是扫平面法的具体实现代码，该函数返回每个像素的最佳视差。创建 stereo.py 文件，将下面的代码添加进去：

```
def plane_sweep_ncc(im_l,im_r,start,steps,wid):
  """ 使用归一化的互相关计算视差图像 """

  m,n = im_l.shape

  # 保存不同求和值的数组
  mean_l = zeros((m,n))
  mean_r = zeros((m,n))
```

```
    s = zeros((m,n))
    s_l = zeros((m,n))
    s_r = zeros((m,n))

    # 保存深度平面的数组
    dmaps = zeros((m,n,steps))

    # 计算图像块的平均值
    filters.uniform_filter(im_l,wid,mean_l)
    filters.uniform_filter(im_r,wid,mean_r)

    # 归一化图像
    norm_l = im_l - mean_l
    norm_r = im_r - mean_r

    # 尝试不同的视差
    for displ in range(steps):
      # 将左边图像移动到右边，计算加和
      filters.uniform_filter(roll(norm_l,-displ-start)*norm_r,wid,s) # 和归一化
      filters.uniform_filter(roll(norm_l,-displ-start)*roll(norm_l,-displ-start),wid,s_l)
      filters.uniform_filter(norm_r*norm_r,wid,s_r) # 和反归一化

      # 保存 ncc 的分数
      dmaps[:,:,displ] = s/sqrt(s_l*s_r)

    # 为每个像素选取最佳深度
    return argmax(dmaps,axis=2)
```

首先，因为 uniform_filter() 函数的输入参数为数组，我们需要创建用于保存滤波结果的一些数组。然后，我们创建一个数组来保存每个平面，从而能够在最后一个纬度上使用 argmax() 函数找到对于每个像素的最佳深度。该函数从 start 偏移出发，在所有的 steps 偏移上迭代。使用 roll() 函数平移一幅图像，然后使用滤波计算 NCC 的三个求和操作。

下面是载入图像，并使用该函数计算偏移图的完整例子：

```
import stereo

im_l = array(Image.open('scene1.row3.col3.ppm').convert('L'),'f')
im_r = array(Image.open('scene1.row3.col4.ppm').convert('L'),'f')

# 开始偏移，并设置步长
steps = 12
start = 4
```

```
# ncc 的宽度
wid = 9

res = stereo.plane_sweep_ncc(im_l,im_r,start,steps,wid)

import scipy.misc
scipy.misc.imsave('depth.png',res)
```

这里首先从经典“tsukuba”数据集中载入一对图像，然后将其灰度化。接下来，设置扫平面函数所需的参数，包括尝试偏移的数目、初始值和 NCC 路径的宽度。你会发现，该方法非常快，至少快于使用 NCC 进行特征匹配。这也是使用滤波器来计算的原因。

该方法同样适用于其他滤波器。均匀滤波器给定正方形图像块中所有像素相同的权值。但是，在一些例子中，其他滤波器对 NCC 的计算可能更为适用。下面是使用高斯滤波器替换均匀滤波器，产生更加平滑视差图的例子。将其添加到 stereo.py 文件中：

```
def plane_sweep_gauss(im_l,im_r,start,steps,wid):
  """ 使用带有高斯加权周边的归一化互相关计算视差图像 """

  m,n = im_l.shape

  # 保存不同加和的数组
  mean_l = zeros((m,n))
  mean_r = zeros((m,n))
  s = zeros((m,n))
  s_l = zeros((m,n))
  s_r = zeros((m,n))

  # 保存深度平面的数组
  dmaps = zeros((m,n,steps))

  # 计算平均值
  filters.gaussian_filter(im_l,wid,0,mean_l)
  filters.gaussian_filter(im_r,wid,0,mean_r)

  # 归一化图像
  norm_l = im_l - mean_l
  norm_r = im_r - mean_r

  # 尝试不同的视差
  for displ in range(steps):
    # 将左边图像移动到右边，计算加和
    filters.gaussian_filter(roll(norm_l,-displ-start)*norm_r,wid,0,s) # 和归一化
```

```
        filters.gaussian_filter(roll(norm_l,-displ-start)*roll(norm_l,-displ-start),wid,
                        0,s_l)
        filters.gaussian_filter(norm_r*norm_r,wid,0,s_r) # 和反归一化

        # 保存 ncc 的分数
        dmaps[:,:,displ] = s/sqrt(s_l*s_r)

    # 为每个像素选取最佳深度
    return argmax(dmaps,axis=2)
```

除了在滤波中使用了额外的参数，该代码和均匀滤波的代码相同。我们需要在 `gaussian_filter()` 函数中传入零参数来表示我们使用的是标准高斯函数，而不是其他任何导数（详见 1.4.2 节）。

你可以像前面的扫平面函数一样来使用该函数。图 5-10 和图 5-11 所示为这两个扫平面实现操作在一些标准立体基准图像上的结果。这些图像来自于参考文献 [29] 和 [30]，可以从 http://vision.middlebury.edu/stereo/data/ 下载。这里我们使用“tsukuba”和“cones”图像，在标准版本中设置 `wid` 为 9，高斯版本中设置 `wid` 为 3。上面一行图像所示为图像对，左下方图像是标准的 NCC 扫平面法重建的视差图，右下方是高斯版本的视差图。可以看到，与标准版本相比，高斯版本具有较少的噪声，但缺少很多细节信息。

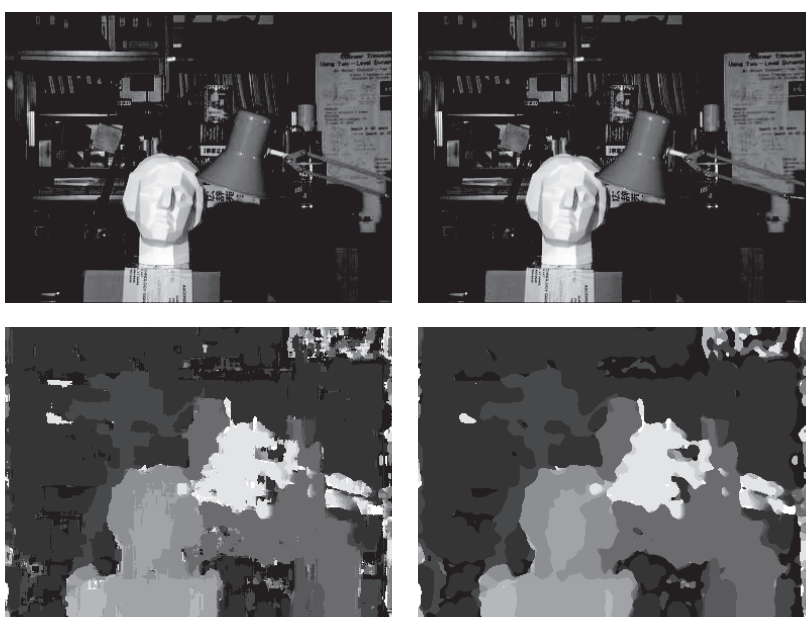


图 5-10：使用归一化互相关从一对立体图像中计算视差图

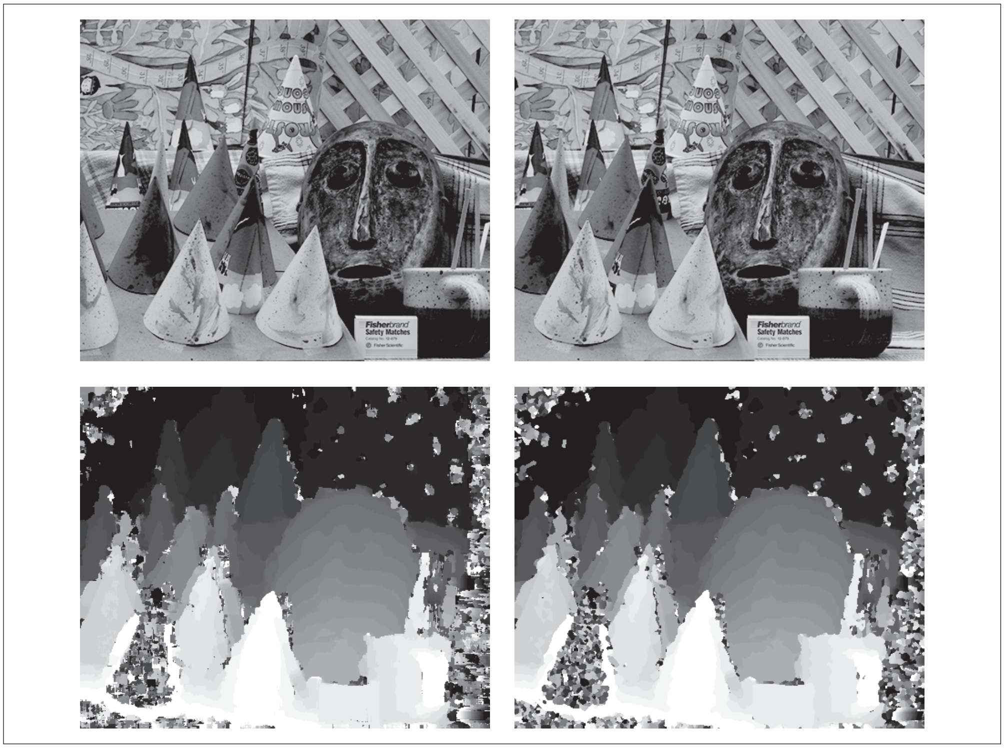


图 5-11：使用归一化互相关从一对立体图像中计算视差图

练习

(1) 使用本章介绍的技术来验证 2.3.2 节中白宫例子的匹配（使用自己的例子更好），试看能否提高结果的性能。

(2) 计算图像对的特征匹配，并估计基础矩阵。使用外极线作为第二个输入，通过在外极线上对每个特征点寻找最佳的匹配来找到更多的匹配。

(3) 制作一个包含三幅或更多图像的数据集。挑选一对图像，计算三维点和照相机矩阵。匹配特征到剩下的图像中以获得对应。然后利用这些对应的三维点使用后方交汇法，计算其他图像的照相机矩阵。绘制这些三维点和照相机的位置。你可以使用自己的数据集，也可以使用牛津多视图数据集的数据集。

(4) 以 NCC 例子中相同的方式使用滤波器，来实现使用 SSD（squared differences，平方差）而不是 NCC 的立体视差图算法版本。

(5) 尝试使用 1.5 节中的 ROF 去噪来对立体深度图进行去噪。对产生锋利边缘噪声的互相关图像块大小进行实验，使用平滑去除产生的噪声。

(6) 一个提高视差图效果的方法是，比较从左图到右图计算出的视差图和从右图到左图计算出的视差图，然后仅保留一致的部分。该操作会清除视差图中闭塞的部分。实现该想法，然后将结果和一个方向的扫平面的结果比较。
(7) 纽约公共图书馆有很多历史悠久的立体照片。打开 http://stereo.nypl.org/gallery 并下载你喜欢的照片（点击右键，保存为 JPEG 格式的文件）。这些图像应该进行了矫正。将照片切成两部分，在该图像上运行致密深度重建代码，然后查看输出结果。

第6章

图像聚类

本章将介绍几种聚类方法，并展示如何利用它们对图像进行聚类，从而寻找相似的图像组。聚类可以用于识别、划分图像数据集，组织与导航。此外，我们还会对聚类后的图像进行相似性可视化。

6.1 K-means聚类

K-means 是一种将输入数据划分成 k 个簇的简单的聚类算法。K-means 反复提炼初始评估的类中心，步骤如下：

(1) 以随机或猜测的方式初始化类中心 $\boldsymbol{u}_i$，i=1⋯k；
(2) 将每个数据点归并到离它距离最近的类中心所属的类 c_i；
(3) 对所有属于该类的数据点求平均，将平均值作为新的类中心；
(4) 重复步骤（2）和步骤（3）直到收敛。

K-means 试图使类内总方差最小：

$$V=\sum_{i=1}^{k}\sum_{\boldsymbol{x}_j\in c_i}(\boldsymbol{x}_j-\boldsymbol{\mu}_i)^2$$

$\boldsymbol{x}_j$ 是输入数据，并且是矢量。该算法是启发式提炼算法，在很多情形下都适用，但是并不能保证得到最优的结果。为了避免初始化类中心时没选取好类中心初值所造成的影响，该算法通常会初始化不同的类中心进行多次运算，然后选择方差 V 最小的结果。

K-means 算法最大的缺陷是必须预先设定聚类数 k，如果选择不恰当则会导致聚类出来的结果很差。其优点是容易实现，可以并行计算，并且对于很多别的问题不需要任何调整就能够直接使用。

6.1.1 SciPy聚类包

尽管 K-means 算法很容易实现，但我们没有必要自己实现它。SciPy 矢量量化包 scipy.cluster.vq 中有 K-means 的实现，下面是使用方法。

为便于说明，我们先生成简单的二维数据：

```
from scipy.cluster.vq import *

class1 = 1.5 * randn(100,2)
class2 = randn(100,2) + array([5,5])
features = vstack((class1,class2))
```

上面的代码生成两类二维正态分布数据。用 k=2 对这些数据进行聚类：

```
centroids,variance = kmeans(features,2)
```

由于 SciPy 中实现的 K-means 会计算若干次（默认为 20 次），并为我们选择方差最小的结果，所以这里返回的方差并不是我们真正需要的。现在，你可以用 SciPy 包中的矢量量化函数对每个数据点进行归类：

```
code,distance = vq(features,centroids)
```

通过上面得到的 code，我们可以检查是否有归类错误。为了将其可视化，我们可以画出这些数据点及最终的聚类中心：

```
figure()
ndx = where(code==0)[0]
plot(features[ndx,0],features[ndx,1],'*')
ndx = where(code==1)[0]
plot(features[ndx,0],features[ndx,1],'r.')
plot(centroids[:,0],centroids[:,1],'go')
axis('off')
show()
```

函数 where() 给出每个类的索引，绘制出的结果如图 6-1 所示。

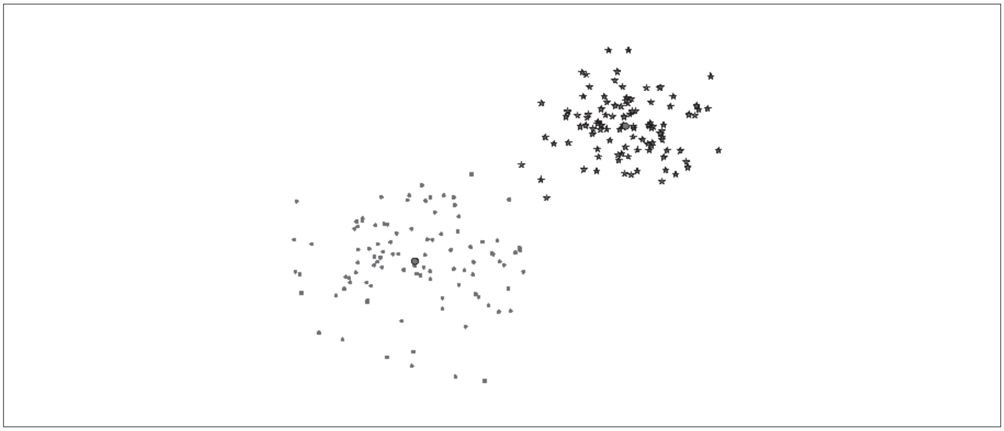


图 6-1：一个对二维数据用 K-means 进行聚类的示例。类中心标记为绿色大圆环，预测出的类分别标记为蓝色星号和红色点。

6.1.2 图像聚类

让我们在 1.3.6 节的字体图像上，我们用 K-means 对这些字体图像进行聚类。文件 selectedfontimages.zip 包含 66 幅来自该字体数据集 fontinages 的图像（为了便于说明这些聚类簇，我们选择这些图像做简单概述）。我们利用之前计算过的前 40 个主成分进行投影，用投影系数作为每幅图像的向量描述符。用 pickle 模块载入模型文件，在主成分上对图像进行投影，然后用下面的方法聚类：

```
import imtools
import pickle
from scipy.cluster.vq import *

# 获取 selected-fontimages 文件下图像文件名，并保存在列表中
imlist = imtools.get_imlist('selected_fontimages/')
imnbr = len(imlist)

# 载入模型文件
with open('a_pca_modes.pkl','rb') as f:
  immean = pickle.load(f)
  V = pickle.load(f)

# 创建矩阵，存储所有拉成一组形式后的图像
immatrix = array([array(Image.open(im)).flatten()
                        for im in imlist],'f')

# 投影到前 40 个主成分上
immean = immean.flatten()
```

```
projected = array([dot(V[:40],immatrix[i]-immean) for i in range(imnbr)])

# 进行 k-means 聚类
projected = whiten(projected)
centroids,distortion = kmeans(projected,4)

code,distance = vq(projected,centroids)
```

与之前一样，上述代码 code 变量中包含的是每幅图像属于哪个簇。这里，我们设定聚类数 k=4，同时用 SciPy 的 whiten() 函数对数据“白化”处理，并进行归一化，使每个特征具有单位方差。你可以试着改变其中的参数，比如主成分数目和 k，观察聚类结果有何改变。利用下面的代码可以可视化聚类后的结果：

```
# 绘制聚类簇
for k in range(4):
    ind = where(code==k)[0]
    figure()
    gray()
    for i in range(minimum(len(ind),40)):
        subplot(4,10,i+1)
        imshow(immatrix[ind[i]].reshape((25,25)))
        axis('off')
show()
```

这里我们将每个簇显示在一个独立图形窗口中，且在该图形窗口中最多可以显示 40 幅图像。我们用 PyLab 的 subplot() 函数来设定网格数，图 6-2 是上面对字体图像聚成 4 类后的可视化结果。

关于 SciPy 中 K-means 实现方法以及 scipy.cluster.vq 模块，详见参考指南：http://docs.scipy.org/doc/scipy/reference/cluster.vq.html.

6.1.3　在主成分上可视化图像

为了便于观察上面是如何利用主成分进行聚类的，我们可以在一对主成分方向的坐标上可视化这些图像。一种方法是将图像投影到两个主成分上，改变投影为：

```
projected = array([dot(V[[0,2]],immatrix[i]-immean) for i in range(imnbr)])
```

以得到相应的坐标（在这里 V[[0,2]] 分别是第一个和第三个主成分）。当然，你也可以将其投影到所有成分上，之后挑选出你需要的列。

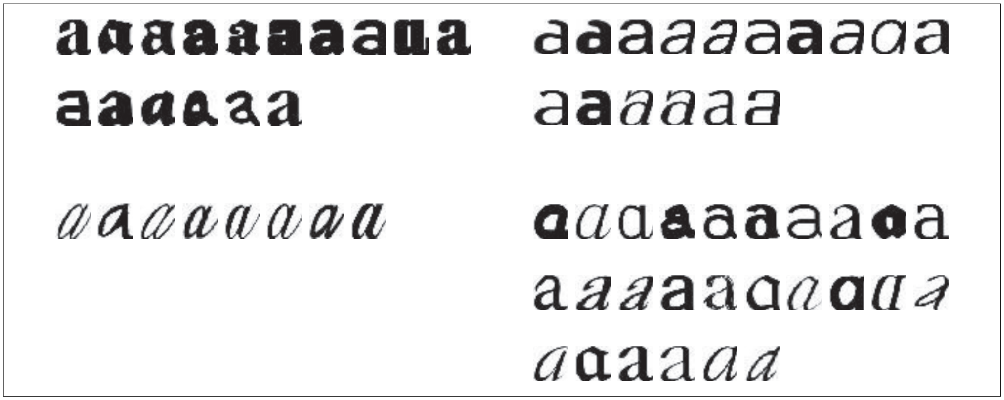


图 6-2：用 40 个主成分数对字体图像进行 K-means 聚类（*k*=4）

我们用 PIL 中的 `ImageDraw` 模块进行可视化。假设你有如上所示投影后的图像，及保存有图像文件名的列表，利用下面简短的脚本可以生成如图 6-3 所示的结果：

```
from PIL import Image, ImageDraw

# 高和宽
h,w = 1200,1200

# 创建一幅白色背景图
img = Image.new('RGB',(w,h),(255,255,255))
draw = ImageDraw.Draw(img)

# 绘制坐标轴
draw.line((0,h/2,w,h/2),fill=(255,0,0))
draw.line((w/2,0,w/2,h),fill=(255,0,0))

# 缩放以适应坐标系
scale = abs(projected).max(0)
scaled = floor(array([ (p / scale) * (w/2-20,h/2-20) + (w/2,h/2) for p in projected]))

# 粘贴每幅图像的缩略图到白色背景图片
for i in range(imnbr):
    nodeim = Image.open(imlist[i])
    nodeim.thumbnail((25,25))
    ns = nodeim.size
    img.paste(nodeim,(scaled[i][0]-ns[0]//2,scaled[i][1]-
        ns[1]//2,scaled[i][0]+ns[0]//2+1,scaled[i][1]+ns[1]//2+1))

img.save('pca_font.jpg')
```

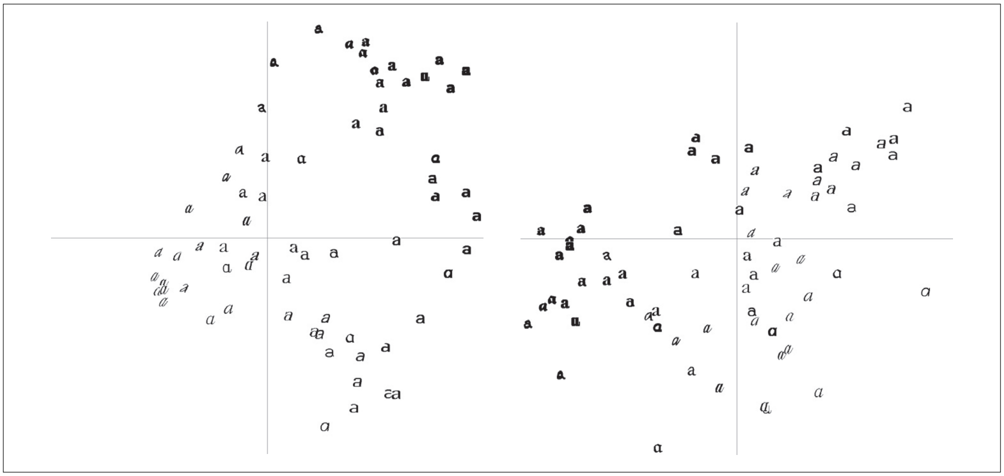


图 6-3：在成对主成分上投影的字体图像。左图用的是第一个和第二个主成分，右图用的是第二个和第三个主成分

这里，我们用到了整数或 floor 向下取整除法运算符 //，通过移去小数点后面的部分，可以返回各个缩略图在白色背景中对应的整数坐标位置。

这类图像说明这些字体图像在 40 维里的分布情况，对于选择一个好的描述子很有帮助。可以很清楚地看到，二维投影后相似的字体图像距离较近。

6.1.4　像素聚类

在结束本节之前，我们来看一个对单幅图像中的像素而非全部图像进行聚类的例子。将图像区域或像素合并成有意义的部分称为图像分割，它是第 9 章的主题。除了在一些简单的图像上，单纯在像素水平上应用 K-means 得出的结果往往是毫无意义的。要产生有意义的结果，往往需要更复杂的类模型而非平均像素色彩或空间一致性。现在，我们仅会在 RGB 三通道的像素值上运用 K-means 进行聚类，分割问题的处理方法会在之后谈到（9.2 节）给予关注及给出细节部分。

下面的代码示例载入一幅图像，用一个步长为 steps 的方形网格在图像中滑动，每滑一次对网格中图像区域像素求平均值，将其作为新生成的低分辨率图像对应位置处的像素值，并用 K-means 进行聚类：

```
from scipy.cluster.vq import *
from scipy.misc import imresize

steps = 50 # 图像被划分成 steps×steps 的区域
im = array(Image.open('empire.jpg'))
```

```
dx = im.shape[0] / steps
dy = im.shape[1] / steps

# 计算每个区域的颜色特征
features = []
for x in range(steps):
  for y in range(steps):
     R = mean(im[x*dx:(x+1)*dx,y*dy:(y+1)*dy,0])
     G = mean(im[x*dx:(x+1)*dx,y*dy:(y+1)*dy,1])
     B = mean(im[x*dx:(x+1)*dx,y*dy:(y+1)*dy,2])
     features.append([R,G,B])
features = array(features,'f') # 变为数组

# 聚类
centroids,variance = kmeans(features,3)
code,distance = vq(features,centroids)

# 用聚类标记创建图像
codeim = code.reshape(steps,steps)
codeim = imresize(codeim,im.shape[:2],interp='nearest')

figure()
imshow(codeim)
show()
```

K-means 的输入是一个有 steps × steps 行的数组，数组的每一行有 3 列，各列分别为区域块 R、G、B 三个通道的像素平均值。为可视化最后的结果，我们用 SciPy 的 imresize() 函数在原图像坐标中显示这幅 steps × steps 的图像。参数 interp 指定插值方法；我们在这里采用最近邻插值法，以便在类间进行变换时不需要引入新的像素值。

图 6-4 显示了用 50 × 50 和 100 × 100 窗口对两幅相对比较简单的示例图像进行像素聚类后的结果。注意，K-means 标签的次序是任意的（在这里的标签指最终结果中图像的颜色）。正如你所看到的，尽管利用窗口对它进行了下采样，但结果仍然是有噪声的。如果图像某些区域没有空间一致性，则很难将它们分开，如下方图中小男孩和草坪的图。空间一致性和更好的分割方法以及其他的图像分割算法会在后面讨论。现在，让我们继续来看下一个基本的聚类算法。



图 6-4：基于颜色像素值用 K-means 对像素进行聚类的结果。左边是原始图像；中间是用 k=3 和 50×50 大小的窗口进行聚类的结果；右边是用 k=3 和 100×100 大小的窗口进行聚类的结果

6.2 层次聚类

层次聚类（或*凝聚式聚类*）是另一种简单但有效的聚类算法，其思想是基于样本间成对距离建立一个简相似性树。该算法首先将特征向量距离最近的两个样本归并为一组，并在树中创建一个“平均”节点，将这两个距离最近的样本作为该“平均”节点下的子节点；然后在剩下的包含任意平均节点的样本中寻找下一个最近的对，重复进行前面的操作。在每一个节点处保存了两个子节点之间的距离。遍历整个树，通过设定的阈值，遍历过程可以在比阈值大的节点位置终止，从而提取出聚类簇。

层次聚类有若干优点。例如，利用树结构可以可视化数据间的关系，并显示这些簇是如何关联的。在树中，一个好的特征向量可以给出一个很好的分离结果。另外一个优点是，对于给定的不同的阈值，可以直接利用原来的树，而不需要重新计算。

不足之处是，对于实际需要的聚类簇，我们需要选择一个合适的阈值。

让我们看看层次聚类算法怎样在代码中体现 [1]。创建文件 hcluster.py，将下面代码添加进去（该代码受参考文献 [31] 中层次聚类例子的启发）：

```
from itertools import combinations

class ClusterNode(object):
  def __init__(self,vec,left,right,distance=0.0,count=1):
    self.left = left
    self.right = right
    self.vec = vec
    self.distance = distance
    self.count = count # 只用于加权平均

  def extract_clusters(self,dist):
    """ 从层次聚类树中提取距离小于 dist 的子树簇群列表 """
    if self.distance < dist:
      return [self]
    return self.left.extract_clusters(dist) + self.right.extract_clusters(dist)

  def get_cluster_elements(self):
    """ 在聚类子树中返回元素的 id """
    return self.left.get_cluster_elements() + self.right.get_cluster_elements()

  def get_height(self):
    """ 返回节点的高度，高度是各分支的和 """
    return self.left.get_height() + self.right.get_height()

  def get_depth(self):
    """ 返回节点的深度，深度是每个子节点取最大再加上它的自身距离 """
    return max(self.left.get_depth(), self.right.get_depth()) + self.distance

class ClusterLeafNode(object):
  def __init__(self,vec,id):
    self.vec = vec
    self.id = id

  def extract_clusters(self,dist):
    return [self]

  def get_cluster_elements(self):
    return [self.id]
```

注 1：在 Scipy 聚类包中，有一个层次聚类的版本，如果你喜欢可以直接使用。因为我们需要创建树、并用缩略图可视化树状图的类，所以这里我们不使用该版本。

```
def get_height(self):
  return 1

def get_depth(self):
  return 0

def L2dist(v1,v2):
  return sqrt(sum((v1-v2)**2))

def L1dist(v1,v2):
  return sum(abs(v1-v2))

def hcluster(features,distfcn=L2dist):
  """ 用层次聚类对行特征进行聚类 """

  # 用于保存计算出的距离
  distances = {}

  # 每行初始化为一个簇
  node = [ClusterLeafNode(array(f),id=i) for i,f in enumerate(features)]

  while len(node)>1:
    closest = float('Inf')

    # 遍历每对，寻找最小距离
    for ni,nj in combinations(node,2):
      if (ni,nj) not in distances:
        distances[ni,nj] = distfcn(ni.vec,nj.vec)

      d = distances[ni,nj]
      if d<closest:
        closest = d
        lowestpair = (ni,nj)
    ni,nj = lowestpair

    # 对两个簇求平均
    new_vec = (ni.vec + nj.vec) / 2.0

    # 创建新的节点
    new_node = ClusterNode(new_vec,left=ni,right=nj,distance=closest)
    node.remove(ni)
    node.remove(nj)
    node.append(new_node)

return node[0]
```

我们为树节点创建了两个类，即 ClusterNode 和 ClusterLeafNode，这两个类将用于创建聚类树，其中函数 hcluster() 用于创建树。首先创建一个包含叶节点的列表，然后根据选择的距离度量方式将距离最近的对归并到一起，返回的终节点即为树的根。对于一个行为特征向量的矩阵，运行 hcluster() 会创建和返回聚类树。

距离度量的选择依赖于实际的特征向量，这里我们利用欧式距离 L_2（同时提供了 L_1 距离度量函数），不过你可以创建任意距离度量函数，并将它作为参数传递给 hcluster()。对于每个子树，计算其所有节点特征向量的平均值，作为新的特征向量来表示该子树，并将每个子树视为一个对象。当然，还有其他将哪两个节点合并在一起的方案，比如在两个子树中使用对象间距离最小的单向锁，及在两个子树中用对象间距离最大的完全锁。选择不同的锁会生成不同类型的聚类树。

为了从树中提取聚类簇，需要从顶部遍历树直至一个距离小于设定阈值的节点终止，这通过递归很容易做到。ClusterNode 的 extract_clusters() 方法用于处理该过程，如果节点间距离小于阈值，则用一个列表返回节点，否则调用子节点（叶节点通常返回它们自身）。调用该函数会返回一个包含聚类簇的子树列表。对于每一个子聚类簇，为了得到包含对象 id 的叶节点，需要遍历每个子树，并用方法 get_cluster_elements() 返回一个包含叶节点的列表。

下面，我们在一个简单的例子中观察该聚类过程。首先创建一些二维数据点（和之前 K-means 一样）：

```
class1 = 1.5 * randn(100,2)
class2 = randn(100,2) + array([5,5])
features = vstack((class1,class2))
```

对这些数据点进行聚类，设定阈值（这里的阈值设定为 5），从列表中提取这些聚类簇，并在控制台打印出来：

```
import hcluster

tree = hcluster.hcluster(features)

clusters = tree.extract_clusters(5)

print 'number of clusters', len(clusters)
for c in clusters:
  print c.get_cluster_elements()
```

打印结果应该与下面类似：

```
number of clusters 2
```

```
[184, 187, 196, 137, 174, 102, 147, 145, 185, 109, 166, 152, 173, 180, 128, 163, 141,
178, 151, 158, 108,182, 112, 199, 100, 119, 132, 195, 105, 159, 140, 171, 191, 164, 130,
149, 150, 157, 176, 135, 123, 131,118, 170, 143, 125, 127, 139, 179, 126, 160, 162, 114,
122, 103, 146, 115, 120, 142, 111, 154, 116, 129,136, 144, 167, 106, 107, 198, 186, 153,
156, 134, 101, 110, 133, 189, 168, 183, 148, 165, 172, 188, 138,192, 104, 124, 113, 194,
190, 161, 175, 121, 197, 177, 193, 169, 117, 155]

[56, 4, 47, 18, 51, 95, 29, 91, 23, 80, 83, 3, 54, 68, 69, 5, 21, 1, 44, 57, 17, 90, 30,
22, 63, 41, 7, 14, 59, 96, 20, 26, 71, 88, 86, 40, 27, 38, 50, 55, 67, 8, 28, 79, 64,
66, 94, 33, 53, 70, 31, 81, 9, 75, 15, 32, 89, 6, 11, 48, 58, 2, 39, 61, 45, 65, 82, 93,
97, 52, 62, 16, 43, 84, 24, 19, 74, 36, 37, 60, 87, 92, 181, 99, 10, 49, 12, 76, 98, 46,
72, 34, 35, 13, 73, 78, 25, 42, 77, 85]
```

理想情况下，你应该得到两一个聚类簇，但是在实际数据中，你可能会得到三类或更多这主要依赖于实际生成的二维数据。在这个对二维数据聚类的简单例子中，一个类中的值应该小于 100，另外一个应该大于等于 100。

图像聚类

我们来看一个基于图像颜色信息对图像进行聚类的例子。文件 sunsets.zip 中包含 100 张图像，这些图像是用“sunset”和“sunsets”关键字在 Flickr 下载下来的。在这个例子中，我们用颜色直方图作为每幅图像的特征向量。虽然这样处理有些简单粗糙，但仍然能够很好地说明分层聚类的过程。在包含这些日落图像的文件夹中运行下面的代码：

```
import os
import hcluster

# 创建图像列表
path = 'flickr-sunsets/'
imlist = [os.path.join(path,f) for f in os.listdir(path) if f.endswith('.jpg')]

# 提取特征向量，每个颜色通道量化成 8 个小区间
features = zeros([len(imlist), 512])
for i,f in enumerate(imlist):
  im = array(Image.open(f))

  # 多维直方图
  h,edges = histogramdd(im.reshape(-1,3),8,normed=True,
                        range=[(0,255),(0,255),(0,255)])
  features[i] = h.flatten()

tree = hcluster.hcluster(features)
```

我们将 R、G、B 三个颜色通道作为特征向量，将其传递到 NumPy 的 histogramdd() 中，该函数能够计算多维直方图（本例中是三维）。我们在每个颜色通道中使用 8 个

小区间进行量化，将三个通道量化后的小区间拉成一行后便可用 512（8×8×8）维的特征向量描述每幅图像。为避免图像尺寸不一致，我们用“normed=True”归一化直方图，并将每个颜色通道范围设置为 0...255。将 reshape() 第一个参数设置为 −1 会自动确定正确的尺寸，故可以创建一个输入数组来计算以 RGB 颜色值为行向量的直方图。

为了可视化聚类树，我们可以画出树状图。树状图是一种显示树布局的图表。在判定给出的描述子向量好坏，以及在特征场合考虑什么是相似的时候，树状图可以提供有用的信息。将下面的代码添加到 hcluster.py 中：

```
from PIL import Image,ImageDraw

def draw_dendrogram(node,imlist,filename='clusters.jpg'):
  """ 绘制聚类树状图，并保存到文件中 """

  # 高和宽
  rows = node.get_height()*20
  cols = 1200

  # 距离缩放因子，以便适应图像宽度
  s = float(cols-150)/node.get_depth()

  # 创建图像，并绘制对象
  im = Image.new('RGB',(cols,rows),(255,255,255))
  draw = ImageDraw.Draw(im)

  # 初始化树开始的线条
  draw.line((0,rows/2,20,rows/2),fill=(0,0,0))

  # 递归地画出节点
  node.draw(draw,20,(rows/2),s,imlist,im)
  im.save(filename)
  im.show()
```

这里，画树状图时对于每个节点用了 draw() 方法，将该方法添加到 ClusterNode 类中：

```
def draw(self,draw,x,y,s,imlist,im):
  """ 用图像缩略图递归地画出叶节点 """

  h1 = int(self.left.get_height()*20 / 2)
  h2 = int(self.right.get_height()*20 /2)
  top = y-(h1+h2)
  bottom = y+(h1+h2)

  # 子节点垂直线
  draw.line((x,top+h1,x,bottom-h2),fill=(0,0,0))
```

```
    # 水平线
    ll = self.distance*s
    draw.line((x,top+h1,x+ll,top+h1),fill=(0,0,0))
    draw.line((x,bottom-h2,x+ll,bottom-h2),fill=(0,0,0))

    # 递归地画左边和右边的子节点
    self.left.draw(draw,x+ll,top+h1,s,imlist,im)
    self.right.draw(draw,x+ll,bottom-h2,s,imlist,im)
```

在画实际图像缩略图时，叶节点有自己的方法，将该方法添加到 ClusterLeafNode 类中：

```
def draw(self,draw,x,y,s,imlist,im):
  nodeim = Image.open(imlist[self.id])
  nodeim.thumbnail([20,20])
  ns = nodeim.size
  im.paste(nodeim,[int(x),int(y-ns[1]//2),int(x+ns[0]),int(y+ns[1]-ns[1]//2)])
```

树状图的高和子部分由距离决定，这些都需要调整，以适应所选择的图像分辨率。随着坐标向下传递到下一级，会递归绘制出这些节点，上述代码用 20×20 像素绘制叶节点的缩略图，使用 get_height() 和 get_depth() 这两个辅助函数可以获得树的高和宽。

树状图可以通过下面的代码绘制，并保存在 sunset.pdf 中：

```
hcluster.draw_dendrogram(tree,imlist,filename='sunset.pdf')
```

图 6-5 展示了日落图像聚类后的树状图。可以看到，树中颜色相似的图像距离较近。图 6-6 中为三个示例簇。可以通过下面的代码提取该例子中的簇：

```
# 设置一些（任意的）阈值以可视化聚类簇
clusters = tree.extract_clusters(0.23*tree.distance)

# 绘制聚类簇中元素超过 3 个的那些图像
for c in clusters:
  elements = c.get_cluster_elements()
  nbr_elements = len(elements)
  if nbr_elements>3:
    figure()
    for p in range(minimum(nbr_elements,20)):
      subplot(4,5,p+1)
      im = array(Image.open(imlist[elements[p]]))
      imshow(im)
      axis('off')
show()
```

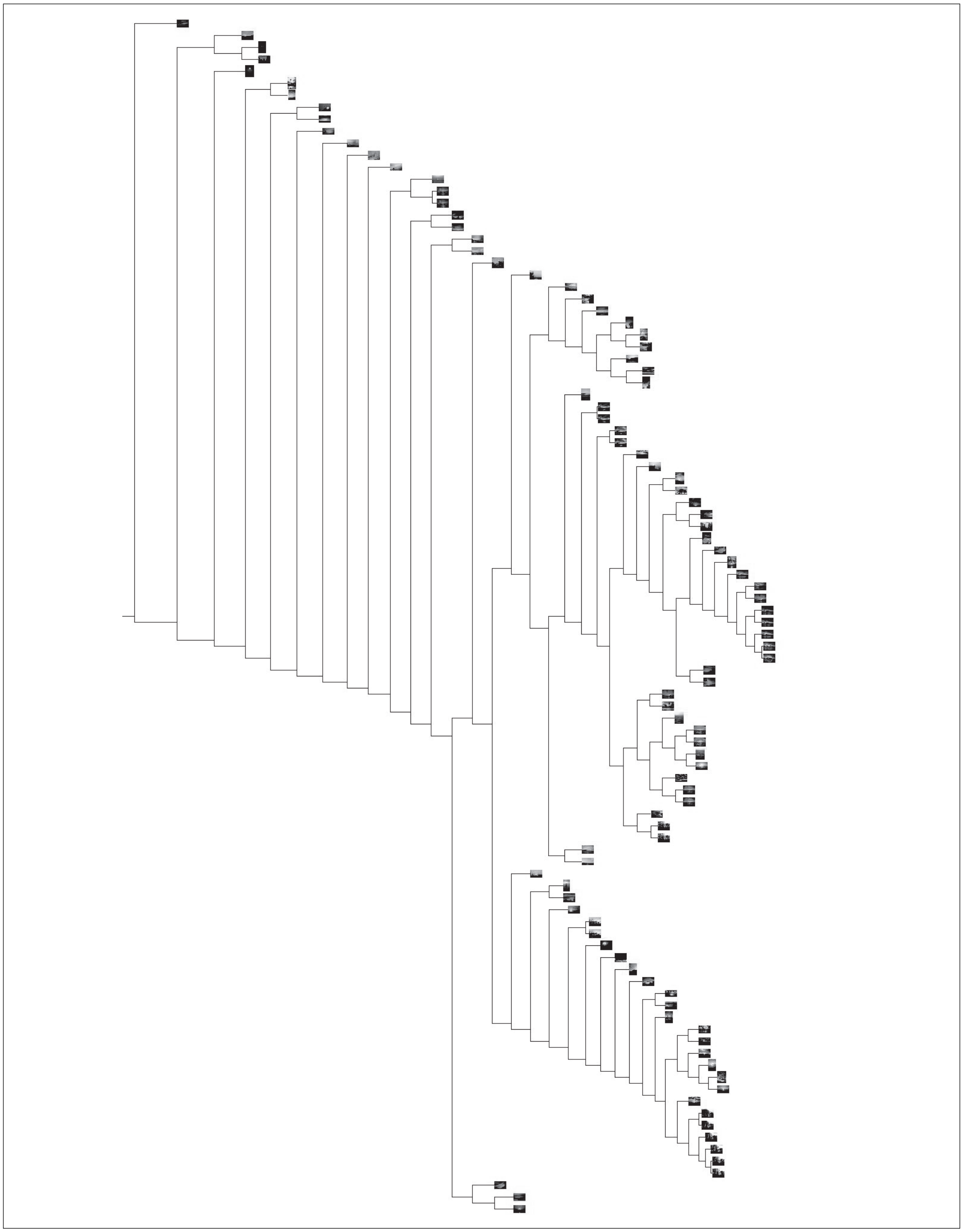


图 6-5：用 100 幅日落图像进行层次聚类，将 RGB 空间的 512 个小区间直方图作为每幅图像的特征向量。树中挨的相近的图像具有相似的颜色分布



图 6-6：用 100 幅日落图像进行层次聚类的示例聚类簇，阈值集合设定为树中最大节点距离的 23%

作为最后一个例子，我们对前面的字体图像创建一个树状图：

```
tree = hcluster.hcluster(projected)
hcluster.draw_dendrogram(tree,imlist,filename='fonts.jpg')
```

其中，projected 和 imlist 是 6.1 节 K-means 例子中的变量。图 6-7 显示了对字体图像进行层次聚类后的树状图。

6.3 谱聚类

谱聚类方法是一种有趣的聚类算法，与前面 K-means 和层次聚类方法截然不同。

对于 n 个元素（如 n 幅图像），相似矩阵（或亲和矩阵，有时也称距离矩阵）是一个 $n \times n$ 的矩阵，矩阵每个元素表示两两之间的相似性分数。谱聚类是由相似性矩阵构建谱矩阵而得名的。对该谱矩阵进行特征分解得到的特征向量可以用于降维，然后聚类。

谱聚类的优点之一是仅需输入相似性矩阵，并且可以采用你所想到的任意度量方式构建该相似性矩阵。像 K-means 和层次聚类需要计算那些特征向量求平均；为了计算平均值，会将特征或描述子限制为向量。而对于谱方法，特征向量就没有类别限制，只要有一个“距离”或“相似性”的概念即可。

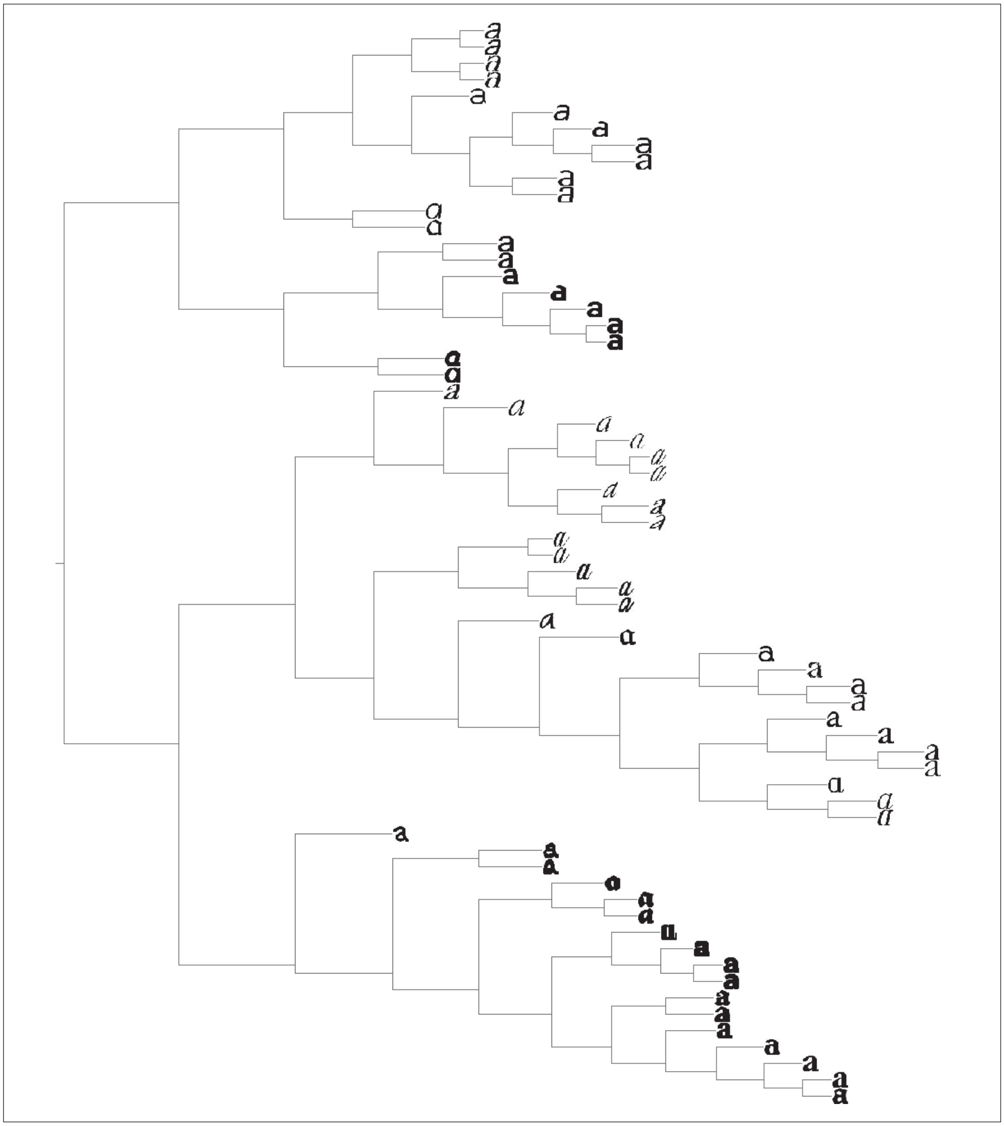


图 6-7：用 66 幅选定字体图像，使用 40 个主成分作为特征量，用层次聚类方法进行聚类

下面说明谱聚类的过程。给定一个 $n \times n$ 的相似矩阵 $\boldsymbol{S}$，s_{ij} 为相似性分数，我们可以创建一个矩阵，称为拉普拉斯矩阵[1]：

$$\boldsymbol{L} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{S}\boldsymbol{D}^{-1/2}$$

注 1：拉普拉斯矩阵有时可以用 $\boldsymbol{L}=\boldsymbol{D}^{1/2}\boldsymbol{S}\boldsymbol{D}^{-1/2}$ 代替，但并无太大差别，因为它只改变特征值，而不改变特征向量。

其中，$\boldsymbol{I}$ 是单位矩阵，$\boldsymbol{D}$ 是对角矩阵，对角线上的元素是 $\boldsymbol{S}$ 对应行元素之和，$D=diag(d_i)$，$d_i=\sum_j s_{ij}$。拉普拉斯矩阵中的 $\boldsymbol{D}^{-1/2}$ 为：

$$\boldsymbol{D}^{-1/2}=\begin{bmatrix} \frac{1}{\sqrt{d_1}} & & & \\ & \frac{1}{\sqrt{d_2}} & & \\ & & \ddots & \\ & & & \frac{1}{\sqrt{d_n}} \end{bmatrix}$$

为了使表述更简洁，对于相似性矩阵中的元素 s_{ij}，我们使用较小的值并且要求 $s_{ij} \geqslant 0$（在这种情况下，距离矩阵可能更合适）。

计算 L 的特征向量，并使用 k 个最大特征值对应的 k 个特征向量，构建出一个特征向量集（记住，我们可能并没有以任何东西来开始），从而可以找到聚类簇。创建一个矩阵，该矩阵的各列是由之前求出的 k 个特征向量构成，每一行可以看做一个新的特征向量，长度为 k。这些新的特征向量可以用诸如 K-means 方法进行聚类，生成最终的聚类簇。本质上，谱聚类算法是将原始空间中的数据转换成更容易聚类的新特征向量。在某些情况下，不会首先使用聚类算法。

讲解了足够的理论，我们来看看真实的例子中谱聚类算法的代码。我们再次使用 1.3.6 节 K-means 例子中的字体图像。

```
from scipy.cluster.vq import *

n = len(projected)

# 计算距离矩阵
S = array([[ sqrt(sum((projected[i]-projected[j])**2))
          for i in range(n) ] for j in range(n)], 'f')

# 创建拉普拉斯矩阵
rowsum = sum(S,axis=0)
D = diag(1 / sqrt(rowsum))
I = identity(n)
L = I - dot(D,dot(S,D))

# 计算矩阵 L 的特征向量
U,sigma,V = linalg.svd(L)

k = 5
# 从矩阵 L 的前 k 个特征向量（eigenvector）中创建特征向量（feature vector）
# 叠加特征向量作为数组的列
```

```
features = array(V[:k]).T

# k-means 聚类
features = whiten(features)
centroids,distortion = kmeans(features,k)
code,distance = vq(features,centroids)

# 绘制聚类簇
for c in range(k):
  ind = where(code==c)[0]
  figure()
  for i in range(minimum(len(ind),39)):
      im = Image.open(path+imlist[ind[i]])
      subplot(4,10,i+1)
      imshow(array(im))
      axis('equal')
      axis('off')
show()
```

在本例中，我们用两两间的欧式距离创建矩阵 ***S***，并对 *k* 个特征向量（eignvector）用常规的 K-means 进行聚类（在该例中，*k*=5）。注意，矩阵 ***V*** 包含的是对特征值进行排序后的特征向量。最后，绘制出这些聚类簇。图 6-8 显示了运行后的聚类簇；需要记住的是，在 K-means 阶段，每次运行的结果可能不同。

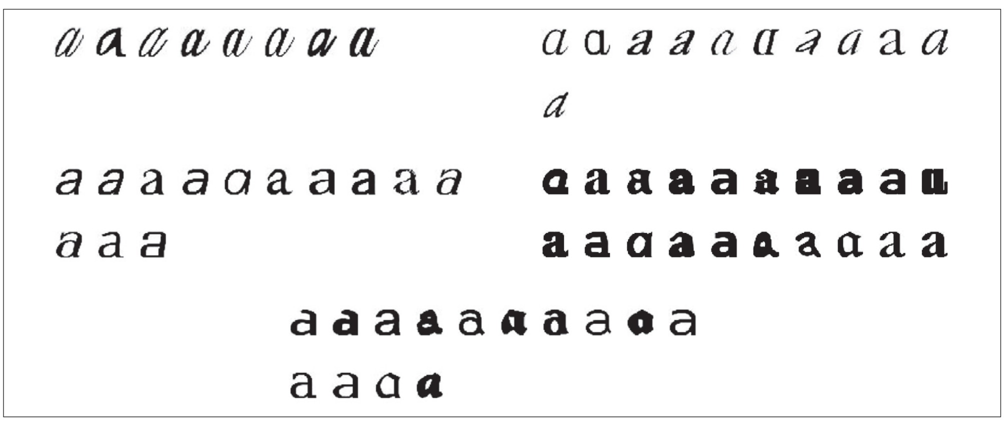


图 6-8：用拉普拉斯矩阵的特征向量对字体图像进行谱聚类

我们也可以在没有任何特征向量或没有严格定义相似性的例子中尝试该算法。2.3 节中带有地理标签的 Panoramio 图像是基于它们之间有多少匹配的局部描述符连接起来的。2.3.2 节的矩阵是一个用分数表示的相似性矩阵，其中分数等于匹配的特征数（没有归一化）。由于 imlist 列表包含了图像文件名，并已用 NumPy 的 savetxt() 将相

似性矩阵保存到了文件中，所以我们只需修改上面代码的前面几行：

```
n = len(imlist)

# 载入相似矩阵并重新格式化
S = loadtxt('panoramio_matches.txt')
S = 1 / (S + 1e-6)
```

这里对分数进行转换，使得相似图像的分数值较小，这样我们就不需要修改上面的代码。我们添加了一个很小的数以防止与 0 相除，后面的代码不需要修改。

在该例中选择 *k* 有些技巧。很多人会认为这里只有两类（即白宫的两侧），以及其他一些垃圾图像。用 *k*=2 可以得到类似图 6-9 的结果，其中一个聚类簇是包含很多白宫一侧的图像，另一个聚类簇是白宫另一侧的图像和其他所有垃圾图像。将 *k* 设定为一个较大的值，比如 *k*=10，则有些聚类簇可能只包含一幅图像（很可能是垃圾图像），另一些是真实的聚类簇。图 6-10 给出了上面示例代码运行的结果。在该例中，仅有两个真实的聚类簇，每个聚类簇包含白宫一个侧面的图像。

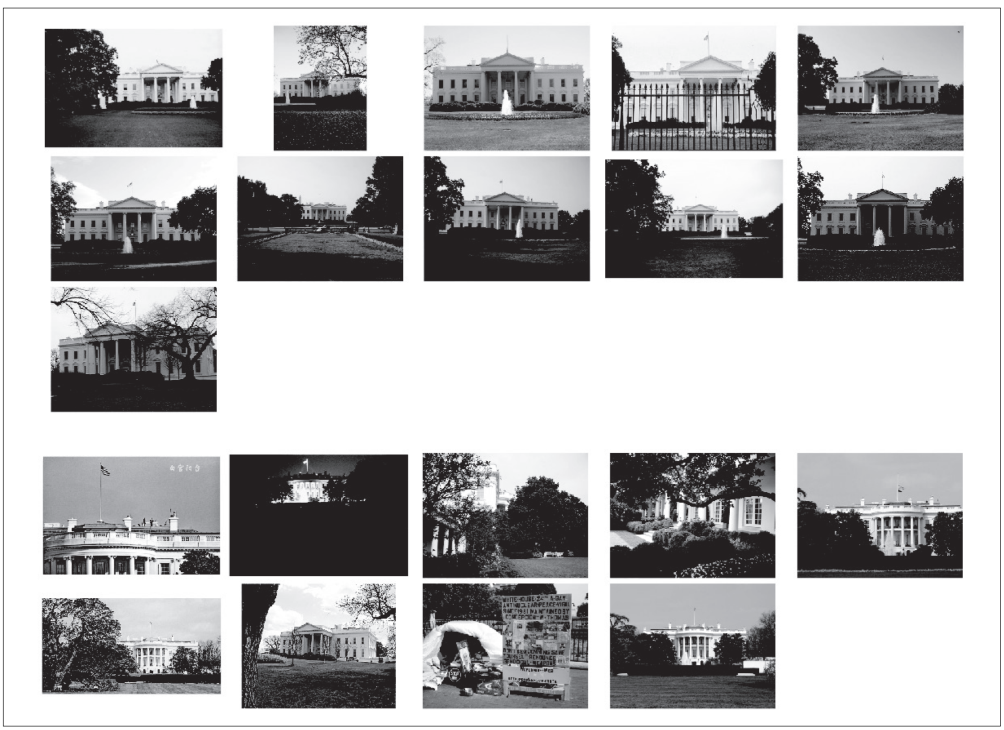


图 6-9：用 *k*=2、局部特征匹配数作为相似性分数对白宫地理图像进行谱聚类的结果

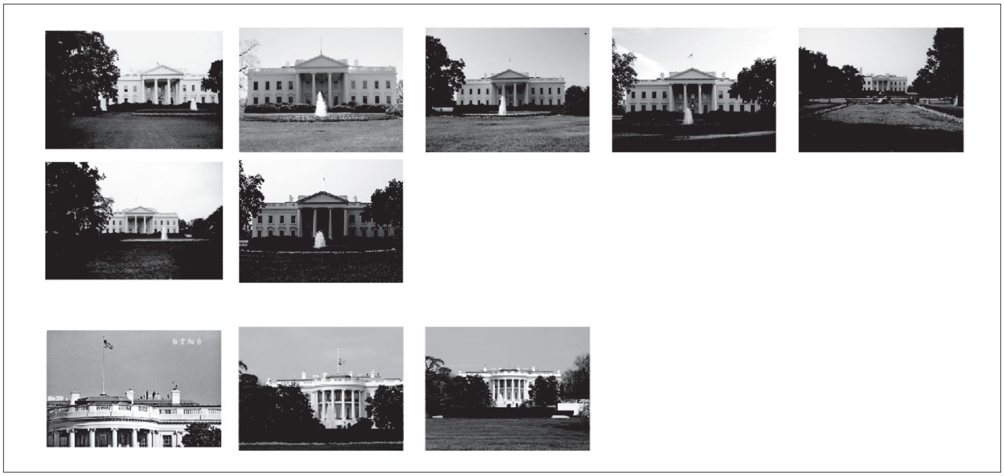


图 6-10：用 k=10，局部特征匹配数作为相似性分数对白宫地理图像进行谱聚类的结果。这里只展示了图像数大于 1 的聚类簇

这里展示的谱聚类算法有很多不同的版本供选择，它们对如何构造矩阵 $\boldsymbol{L}$ 和如何处理特征向量有各自不同的思想。关于谱聚类及一些常用算法的细节，参见综述文章 [37]。

练习

(1) *层次* K-means 是一种聚类方法，该方法递归地应用 K-means 进行聚类，创建一棵逐步提炼聚类簇的树。在此情形下，树的每一个节点都有 k 个子节点。实现层次 K-means 算法并应用到前面的字体图像上。

(2) 类似层次聚类生成树状图，使用上面练习中的层次 K-means 算法，将树可视化，显示每个聚类簇节点的平均图像。提示：你可以获取平均 PCA 系数特征向量，并用 PCA 的基对每个特征向量进行图像合成。

(3) 通过修改层次聚类所使用的类以包含节点下的图像数，你可以获得一种简单快速的方法来寻找给定大小的相似组。实现这个小的改动，并用于处理真实的数据，看看其表现如何。

(4) 用单向锁和完全锁来构建层次聚类树进行实验，这些聚类后的簇有什么不同？

(5) 在一些谱聚类算法中，用到的是矩阵是 $\boldsymbol{D}^{-1}\boldsymbol{S}$ 而不是 $\boldsymbol{L}$。用该矩阵代替拉普拉斯矩阵，并将其应用于一些不同的数据集。

(6) 从 Flickr 下载一些用不同关键字搜索出的图像，像之前的日落图像一样提取 RGB 直方图，用本章介绍的某个聚类方法对这些图像进行聚类。你能用这些聚类簇分开这些图像吗？

第 7 章

图像搜索

本章将展示如何利用文本挖掘技术对基于图像视觉内容进行图像搜索。本章阐明了提出利用视觉单词的基本思想，并解释了完整的安装细节，还在一个示例数据集上进行了测试。

7.1 基于内容的图像检索

在大型图像数据库上，CBIR（Content-Based Image Retrieval，基于内容的图像检索）技术用于检索在视觉上具相似性的图像。这样返回的图像可以是颜色相似、纹理相似、图像中的物体或场景相似；总之，基本上可以是这些图像自身共有的任何信息。

对于高层查询，比如寻找相似的物体，将查询图像与数据库中所有的图像进行完全比较（比如用特征匹配）往往是不可行的。在数据库很大的情况下，这样的查询方式会耗费过多时间。在过去的几年里，研究者成功地引入文本挖掘技术到 CBIR 中处理问题，使在数百万图像中搜索具有相似内容的图像成为可能。

从文本挖掘中获取灵感——矢量空间模型

矢量空间模型是一个用于表示和搜索文本文档的模型。我们将看到，它基本上可以应用于任何对象类型，包括图像。该名字来源于用矢量来表示文本文档，这些矢量是由文本词频直方图构成的[1]。换句话说，矢量包含了每个单词出现的次数，而且在

注 1：经常可以看到用“术语”替代“词”，两者在矢量空间模型中表达的意义相同。

其他别的地方包含很多 0 元素。由于其忽略了单词出现的顺序及位置，该模型也被称为 BOW 表示模型。

通过单词计数来构建文档直方图向量 $\boldsymbol{v}$，从而建立文档索引。通常，在单词计数时会忽略掉一些常用词，如“这”“和”“是”等，这些常用词称为*停用词*。由于每篇文档长度不同，故除以直方图总和将向量归一化成单位长度。对于直方图向量中的每个元素，一般根据每个单词的重要性来赋予相应的权重。通常，数据集（或语料库）中一个单词的重要性与它在文档中出现的次数成正比，而与它在语料库中出现的次数成反比。

最常用的权重是 tf-idf（term frequency-inverse document frequency，词频 – 逆向文档频率），单词 w 在文档 d 中的*词频*是：

$$\mathrm{tf}_{w,d} = \frac{n_w}{\sum_j n_j}$$

n_w 是单词 w 在文档 d 中出现的次数。为了归一化，将 n_w 除以整个文档中单词的总数。

逆向文档频率为：

$$\mathrm{idf}_{w,d} = \log \frac{|(D)|}{|\{d: w \in d\}|}$$

$|\boldsymbol{D}|$ 是在语料库 $\boldsymbol{D}$ 中文档的数目，分母是语料库中包含单词 w 的文档数 d。将两者相乘可以得到矢量 $\boldsymbol{v}$ 中对应元素的 tf-idf 权重。关于 tf-idf，详见 http://en.wikipedia.org/wiki/Tf-idf。

上面就是我们目前需要掌握的知识。接下来让我们看看怎样将该模型应用到基于视觉内容的图像索引及搜索中。

7.2 视觉单词

为了将文本挖掘技术应用到图像中，我们首先需要建立视觉等效单词；这通常可以采用 2.2 节中介绍的 SIFT 局部描述子做到。它的思想是将描述子空间量化成一些典型实例，并将图像中的每个描述子指派到其中的某个实例中。这些典型实例可以通过分析训练图像集确定，并被视为视觉单词。所有这些视觉单词构成的集合称为*视觉词汇*，有时也称为*视觉码本*。对于给定的问题、图像类型，或在通常情况下仅需呈现视觉内容，可以创建特定的词汇。

从一个（很大的训练图像）集提取特征描述子，利用一些聚类算法可以构建出视觉

单词。聚类算法中最常用的是 K-means[1]，这里也将采用 K-means。视觉单词并不高端，只是在给定特征描述子空间中的一组向量集，在采用 K-means 进行聚类时得到的视觉单词是聚类质心。用视觉单词直方图来表示图像，则该模型便称为 BOW 模型。

我们首先介绍一个示例数据集，并利用它来说明 BOW 概念。文件 first1000.zip 包含了有肯塔基大学物体识别数据集（或称“ukbench”）的前 1000 幅图像。完整的数据集、公布的基准和一些配套代码参见 http://www.vis.uky.edu/~stewe/ukbench/。该 ukbench 数据集有很多子集，每个子集包含四幅图像，这四幅图像具有相同的场景或物体，而且存储的文件名是连续的，即 0 . . . 3 属于同一图像子集，4 . . . 7 属于另外同一图像子集，以此类推。图 7-1 展示了数据集中的一些图像，附录 B.4 给出了该数据集的细节以及获取方法。



图 7-1：ukbench（肯塔基大学物体识别数据集）数据集中的一些图像

注 1：或在更加高级的场合下使用层次 K-means。

创建词汇

为创建视觉单词词汇，首先需要提取特征描述子。这里，我们使用 SIFT 特征描述子。如前面一样，imlist 包含的是图像的文件名。运行下面的代码，可以得到每幅图像提取出的描述子，并将每幅图像的描述子保存在一个文件中：

```
nbr_images = len(imlist)
featlist = [ imlist[i][:-3]+'sift' for i in range(nbr_images)]

for i in range(nbr_images):
  sift.process_image(imlist[i],featlist[i])
```

创建名为 vocabulary.py 的文件，将下面代码添加进去。该代码创建一个词汇类，以及在训练图像数据集上训练出一个词汇的方法：

```
from scipy.cluster.vq import *
import vlfeat as sift

class Vocabulary(object):

  def __init__(self,name):
    self.name = name
    self.voc = []
    self.idf = []
    self.trainingdata = []
    self.nbr_words = 0
  def train(self,featurefiles,k=100,subsampling=10):
    """ 用含有 k 个单词的 K-means 列出在 featurefiles 中的特征文件训练出一个词汇。对训练数据下
      采样可以加快训练速度 """

    nbr_images = len(featurefiles)
    # 从文件中读取特征
    descr = []
    descr.append(sift.read_features_from_file(featurefiles[0])[1])
    descriptors = descr[0] # 将所有的特征并在一起，以便后面进行 K-means 聚类
    for i in arange(1,nbr_images):
      descr.append(sift.read_features_from_file(featurefiles[i])[1])
      descriptors = vstack((descriptors,descr[i]))

    # K-means: 最后一个参数决定运行次数
    self.voc,distortion = kmeans(descriptors[::subsampling,:],k,1)
    self.nbr_words = self.voc.shape[0]

    # 遍历所有的训练图像，并投影到词汇上
    imwords = zeros((nbr_images,self.nbr_words))
```

```
    for i in range( nbr_images ):
      imwords[i] = self.project(descr[i])

    nbr_occurences = sum( (imwords > 0)*1 ,axis=0)

    self.idf = log( (1.0*nbr_images) / (1.0*nbr_occurences+1) )
    self.trainingdata = featurefiles
  def project(self,descriptors):
    """ 将描述子投影到词汇上，以创建单词直方图 """

    # 图像单词直方图
    imhist = zeros((self.nbr_words))
    words,distance = vq(descriptors,self.voc)
    for w in words:
      imhist[w] += 1

    return imhist
```

Vocabulary 类包含了一个由单词聚类中心 VOC 与每个单词对应的逆向文档频率构成的向量，为了在某些图像集上训练词汇，train() 方法获取包含有 .sift 描后缀的述子文件列表和词汇单词数 k。在 K-means 聚类阶段可以对训练数据下采样，因为如果使用过多特征，会耗费很长时间。

现在在你计算机的某个文件夹中，保存了图像及提取出来的 sift 特征文件，下面的代码会创建一个长为 $k \approx 1000$ 的词汇表。这里，再次假设 imlist 是一个包含了图像文件名的列表：

```
import pickle
import vocabulary

nbr_images = len(imlist)
featlist = [ imlist[i][:-3]+'sift' for i in range(nbr_images) ]

voc = vocabulary.Vocabulary('ukbenchtest')
voc.train(featlist,1000,10)

# 保存词汇
with open('vocabulary.pkl', 'wb') as f:
  pickle.dump(voc,f)
print 'vocabulary is:', voc.name, voc.nbr_words
```

代码最后部分用 pickle 模块保存了整个词汇对象以便后面使用。

7.3 图像索引

在开始搜索之前，我们需要建立图像数据库和图像的视觉单词表示。

7.3.1 建立数据库

在索引图像前，我们需要建立一个数据库。这里，对图像进行索引就是从这些图像中提取描述子，利用词汇将描述子转换成视觉单词，并保存视觉单词及对应图像的单词直方图。从而可以利用图像对数据库进行查询，并返回相似的图像作为搜索结果。

这里，我们使用 SQLite 作为数据库。SQLite 将所有信息都保存到一个文件，是一个易于安装和使用的数据库。由于不涉及数据库和服务器的配置及其他超出本书范围的细节，它很容易上手。SQLite 对应的 Python 版本是 pysqlite，可以从 http://code.google.com/p/pysqlite/ 获取，或在 Mac 和 Linux 系统中通过软件源获取。SQLite 使用 SQL 查询语言，所以如果想用别的数据库，这个转换过程非常简单。

在开始之前，我们需要创建表、索引和索引器 Indexer 类，以便将图像数据写入数据库。首先，创建一个名为 imagesearch.py 的文件，将下面的代码添加进去：

```
import pickle
from pysqlite2 import dbapi2 as sqlite

class Indexer(object):

  def __init__(self,db,voc):
    """ 初始化数据库的名称及词汇对象 """

    self.con = sqlite.connect(db)
    self.voc = voc

  def __del__(self):
    self.con.close()

  def db_commit(self):
    self.con.commit()
```

首先，我们需要用 pickle 模块将这些数组编码成字符串以及将字符串进行解码；SQLite 可以从 pysqlite2 模块中导入（安装细节参见附录 A）。Indexer 类连接数据库，并且一旦创建（调用 __init__() 方法）后就可以保存词汇对象。__del__() 方法可以确保关闭数据库连接，db_commit() 可以将更改写入数据库文件。

我们仅需一个包含三个表单的简单数据库模式。表单 imlist 包含所有要索引的图像

文件名；imwords 包含了一个那些单词的单词索引、用到了哪个词汇、以及单词出现在哪些图像中；最后，imhistograms 包含了全部每幅图像的单词直方图。根据矢量空间模型，我们需要这些以便进行图像比较。表 7-1 展示了该模式。

表7-1：一个用于存储图像及视觉单词的简单数据库模式

imlist	imwords	imhistograms
rowid	imid	imid
filename	wordid	histogram
	vocname	vocname

下面 Indexer 类中的方法用于创建表单及一些有用的索引以加快搜索速度：

```
def create_tables(self):
  """ 创建数据库表单 """

  self.con.execute('create table imlist(filename)')
  self.con.execute('create table imwords(imid,wordid,vocname)')
  self.con.execute('create table imhistograms(imid,histogram,vocname)')
  self.con.execute('create index im_idx on imlist(filename)')
  self.con.execute('create index wordid_idx on imwords(wordid)')
  self.con.execute('create index imid_idx on imwords(imid)')
  self.con.execute('create index imidhist_idx on imhistograms(imid)')
  self.db_commit()
```

7.3.2 添加图像

有了数据库表单，我们便可以在索引中添加图像。为了实现该功能，我们需要在 Indexer 类中添加 add_to_index() 方法。将下面的方法添加到 imagesearch.py 中：

```
def add_to_index(self,imname,descr):
  """ 获取一幅带有特征描述子的图像，投影到词汇上并添加进数据库 """

  if self.is_indexed(imname): return
  print 'indexing', imname

  # 获取图像 id
  imid = self.get_id(imname)

  # 获取单词
  imwords = self.voc.project(descr)
  nbr_words = imwords.shape[0]

  # 将每个单词与图像链接起来
  for i in range(nbr_words):
```

```
      word = imwords[i]
      # wordid 就是单词本身的数字
      self.con.execute("insert into imwords(imid,wordid,vocname)
          values (?,?,?)", (imid,word,self.voc.name))

    # 存储图像的单词直方图
    # 用 pickle 模块将 NumPy 数组编码成字符串
    self.con.execute("insert into imhistograms(imid,histogram,vocname)
          values (?,?,?)", (imid,pickle.dumps(imwords),self.voc.name))
```

该方法获取图像文件名与 Numpy 数组，该数组包含的是在图像找到的描述子。这些描述子投影到词汇上，并插入到 imwords（逐字）和 imhistograms 表单中。我们使用两个辅助函数：is_indxed() 用来检查图像是否已经被索引，get_id() 则对一幅图像文件名给定 id 号。将下面的代码添加进 imagesearch.py：

```
  def is_indexed(self,imname):
    """ 如果图像名字（imname）被索引到，就返回 True"""

    im = self.con.execute("select rowid from imlist where
          filename='%s'" % imname).fetchone()
    return im != None

  def get_id(self,imname):
    """ 获取图像 id，如果不存在，就进行添加 """

    cur = self.con.execute(
    "select rowid from imlist where filename='%s'" % imname)
    res=cur.fetchone()
    if res==None:
      cur = self.con.execute(
      "insert into imlist(filename) values ('%s')" % imname)
      return cur.lastrowid
    else:
      return res[0]
```

你是否注意到我们在 add_to_index() 方法中用到了 Pickle 模块？由于 SQLite 的数据库在存储对象或数组时并没有一个标准类型。所以，我们用 Pickle 的 dumps() 函数创建一个字符串表示，并将其写入数据库。因此，从数据库读取数据时，我们需要拆封该字符串，这在下一节有详细介绍。

下面的示例代码会遍历整个 ukbench 数据库中的样本图像，并将其加入我们的索引。这里，假设列表 imlist 和 featlist 分别包含之前图像文件名及图像描述子，vocabulary.pkl 包含已经训练好的词汇：

```
import pickle
import sift
import imagesearch

nbr_images = len(imlist)

# 载入词汇
with open('vocabulary.pkl', 'rb') as f:
  voc = pickle.load(f)

# 创建索引器
indx = imagesearch.Indexer('test.db',voc)
indx.create_tables()

# 遍历整个图像库，将特征投影到词汇上并添加到索引中
for i in range(nbr_images)[:1000]:
  locs,descr = sift.read_features_from_file(featlist[i])
  indx.add_to_index(imlist[i],descr)

# 提交到数据库
indx.db_commit()
```

现在我们可以检查数据库中的内容了：

```
from pysqlite2 import dbapi2 as sqlite
con = sqlite.connect('test.db')
print con.execute('select count (filename) from imlist').fetchone()
print con.execute('select * from imlist').fetchone()
```

控制台打印结果如下：

```
(1000,)
(u'ukbench00000.jpg',)
```

如果你在最后一行用 `fetchall()` 来代替 `fetchone()`，会得到一个包含所有文件名的长列表。

7.4　在数据库中搜索图像

建立好图像的索引，我们就可以在数据库中搜索相似的图像了。这里，我们用 BoW（Bag-of-Word，词袋模型）来表示整个图像，不过这里介绍的过程是通用的，可以应用于寻找相似的物体、相似的脸、相似的颜色等，它完全取决于图像及所用的描述子。

为实现搜索，我们在 imagesearch.py 中添加 `Searcher` 类：

```
class Searcher(object):

  def __init__(self,db,voc):
    """ 初始化数据库的名称 """
    self.con = sqlite.connect(db)
    self.voc = voc

  def __del__(self):
    self.con.close()
```

一个新的 Searcher 对象连接到数据库，一旦删除便关闭连接，这与之前的 Indexer 类中的处理过程相同。

如果图像数据库很大，逐一比较整个数据库中的所有直方图往往是不可行的。我们需要找到一个大小合理的候选集（这里的“合理”是通过搜索响应时间、所需内存等确定的），单词索引的作用便在于此：我们可以利用单词索引获得候选集，然后只需在候选集上进行逐一比较。

7.4.1 利用索引获取候选图像

我们可以利用建立起来的索引找到包含特定单词的所有图像，这不过是对数据库做一次简单的查询。在 Searcher 类中加入 candidates_from_word() 方法：

```
def candidates_from_word(self,imword):
  """ G 获取包含 imword 的图像列表 """

  im_ids = self.con.execute(
      "select distinct imid from imwords where wordid=%d" % imword).fetchall()
  return [i[0] for i in im_ids]
```

上面会给出包含特定单词的所有图像 id 号。为了获得包含多个单词的候选图像，例如一个单词直方图中的全部非零元素，我们在每个单词上进行遍历，得到包含该单词的图像，并合并这些列表[1]。这里，我们仍然需要在合并了的列表中对每一个图像 id 出现的次数进行跟踪，因为这可以显示有多少单词与单词直方图中的单词匹配。该过程可以通过下面的 candidates_from_histogram 方法完成：

```
def candidates_from_histogram(self,imwords):
  """ 获取具有相似单词的图像列表 """

  # 获取单词 id
  words = imwords.nonzero()[0]
```

注 1：如果不想使用所有单词，你可以根据其倒排文档频率权重进行排序，并使用那些权重最高的单词。

```
# 寻找候选图像
candidates = []
for word in words:
  c = self.candidates_from_word(word)
  candidates+=c

# 获取所有唯一的单词，并按出现次数反向排序
tmp = [(w,candidates.count(w)) for w in set(candidates)]
tmp.sort(cmp=lambda x,y:cmp(x[1],y[1]))
tmp.reverse()

# 返回排序后的列表，最匹配的排在最前面
return [w[0] for w in tmp]
```

该方法从图像单词直方图的非零项创建单词 id 列表，检索每个单词获得候选集并将其合并到 candidates 列表中，然后创建一个元组列表每个元组由单词 id 和次数 count 构成，其中次数 count 是候选列表中每个单词出现的次数。同时，我们还以元组中的第二个元素为准，用 sort() 方法和一个自定义的比较函数对列表进行排序（考虑到后面的效率）。该自定义比较函数进行用 lambda 函数内联声明，对于单行函数声明，使用 lambda 函数非常方便。最后结果返回一个包含图像 id 的列表，排在列表最前面的是最好的匹配图像。

思考下面的例子：

```
src = imagesearch.Searcher('test.db', voc)
locs,descr = sift.read_features_from_file(featlist[0])
iw = voc.project(descr)

print 'ask using a histogram...'
print src.candidates_from_histogram(iw)[:10]
```

该例打印了从索引中查找出的前 10 个图像 id，结果如下（该结果根据你使用的词汇有所不同）：

```
ask using a histogram...
[655, 656, 654, 44, 9, 653, 42, 43, 41, 12]
```

查找出来的前 10 个候选图像都是不准确的。不用担心，我们现在可以取出该列表中任意数量的元素并比较它们的直方图。你将会看到，这可以极大地提高检索效率。

7.4.2 用一幅图像进行查询

利用一幅图像进行查询时，没有必要进行完全的搜索。为了比较单词直方图，Searcher 类需要从数据库读入图像的单词直方图。将下面的方法添加到 Searcher 类中：

```
def get_imhistogram(self,imname):
    """ 返回一幅图像的单词直方图 """

    im_id = self.con.execute(
        "select rowid from imlist where filename='%s'" % imname).fetchone()
    s = self.con.execute(
        "select histogram from imhistograms where rowid='%d'" % im_id).fetchone()

    # 用 pickle 模块从字符串解码 Numpy 数组
    return pickle.loads(str(s[0]))
```

这里，为了在字符串和 NumPy 数组间进行转换，我们再次用到了 pickle 模块，这次使用的是 loads()。

现在，我们可以全部合并到查询方法中：

```
def query(self,imname):
    """ 查找所有与 imname 匹配的图像列表 """

    h = self.get_imhistogram(imname)
    candidates = self.candidates_from_histogram(h)

    matchscores = []
    for imid in candidates:
        # 获取名字
        cand_name = self.con.execute(
            "select filename from imlist where rowid=%d" % imid).fetchone()
        cand_h = self.get_imhistogram(cand_name)
        cand_dist = sqrt( sum(self.voc.idf* (h-cand_h)2 ) ) # 用 L2 距离度量相似性
        matchscores.append( (cand_dist,imid) )

    # 返回排序后的距离及对应数据库 ids 列表
    matchscores.sort()
    return matchscores
```

该 query() 方法获取图像的文件名，检索其单词直方图及候选图像列表（如果你的数据集很大，候选集的大小应该限制在某个最大值）。对于每个候选图像，我们用标准的欧式距离比较它和查询图像间的直方图，并返回一个经排序的包含距离及图像 id 的元组列表。

我们尝试对前一节的图像进行查询：

```
src = imagesearch.Searcher('test.db', voc)
print 'try a query...'
print src.query(imlist[0])[:10]
```

这会再次打印前 10 个结果，包括候选图像与查询图像间的距离，结果应该和下面类似：

```
try a query...
[(0.0, 1), (100.03999200319841, 2), (105.45141061171255, 3), (129.47200469599596, 708),
(129.73819792181484, 707), (132.68006632497588, 4), (139.89639023220005, 10),
(142.31654858097141, 706), (148.1924424523734, 716), (148.22955170950223, 663)]
```

这次结果比前一节中打印出来的 10 个结果要好很多。距离为 0 的图像对应查询图像本身；三幅与查询图像具有相同场景的图像有两幅在除查询图像本身外的前两个位置，第三幅则出现在第五个位置。

7.4.3 确定对比基准并绘制结果

为了评价搜索结果的好坏，我们可以计算前 4 个位置中搜索到相似图像数。这是在 ukbench 图像集上评价搜索性能常采用的评价方式。这里给出了计算分数的函数，将它添加到 imagesearch.py 中，你就可以开始优化查询了：

```
def compute_ukbench_score(src,imlist):
    """ 对查询返回的前 4 个结果计算平均相似图像数，并返回结果 """

    nbr_images = len(imlist)
    pos = zeros((nbr_images,4))
    # 获取每幅查询图像的前 4 个结果
    for i in range(nbr_images):
        pos[i] = [w[1]-1 for w in src.query(imlist[i])[:4]]

    # 计算分数，并返回平均分数
    score = array([ (pos[i]//4)==(i//4) for i in range(nbr_images)])*1.0
    return sum(score) / (nbr_images)
```

该函数获得搜索的前 4 个结果，将 `query()` 返回的索引减去 1，因为数据库索引是从 1 开始的，而图像列表的索引是从 0 开始的。然后，利用每 4 幅图像为一组时相似图像文件名是连续的这一事实，我们用整数相除计算得到最终的分数。分数为 4 时结果最理想；没有一个是准确的，分数为 0；仅检索到相同图像时，分数为 1；找到相同的图像并且其他三个中的两个相同时，分数为 3。

试试下面的代码：

```
imagesearch.compute_ukbench_score(src,imlist)
```

进行 1000 次查询需要耗费较长时间，如果你不想等太久，可以将查询集改为上面查询集的子集：

```
imagesearch.compute_ukbench_score(src,imlist[:100])
```

当得到的分数接近 3 时，我们可以认为结果很好。目前，ukbench 网站给出的最好结果刚刚超过 3；不过需要注意的是，他们用了更多的图像，所以在大数据集上，你在上面所得到的分数会下降。

最后，用于显示实际搜索结果的函数十分有用。添加该函数到 imagesearch.py 中：

```
def plot_results(src,res):
    """ 显示在列表 res 中的图像 """

    figure()
    nbr_results = len(res)
    for i in range(nbr_results):
        imname = src.get_filename(res[i])
        subplot(1,nbr_results,i+1)
        imshow(array(Image.open(imname)))
        axis('off')
    show()
```

对于列表 res 中的任意搜索结果数，都可以调用该函数。例子如下：

```
nbr_results = 6
res = [w[1] for w in src.query(imlist[0])[:nbr_results]]
imagesearch.plot_results(src,res)
```

定义辅助函数：

```
def get_filename(self,imid):
    """ 返回图像 id 对应的文件名 """

    s = self.con.execute(
        "select filename from imlist where rowid='%d'" % imid).fetchone()
    return s[0]
```

它可以将图像的 id 转换为图像文件名，以便在显示搜索结果时载入图像。图 7-2 显示了用 plot_results() 在我们的数据集上进行的一些查询实例。

7.5　使用几何特性对结果排序

让我们简要地看一种用 BoW 模型改进检索结果的常用方法。BoW 模型的一个主要缺点是在用视觉单词表示图像时不包含图像特征的位置信息，这是为获取速度和可伸缩性而付出的代价。

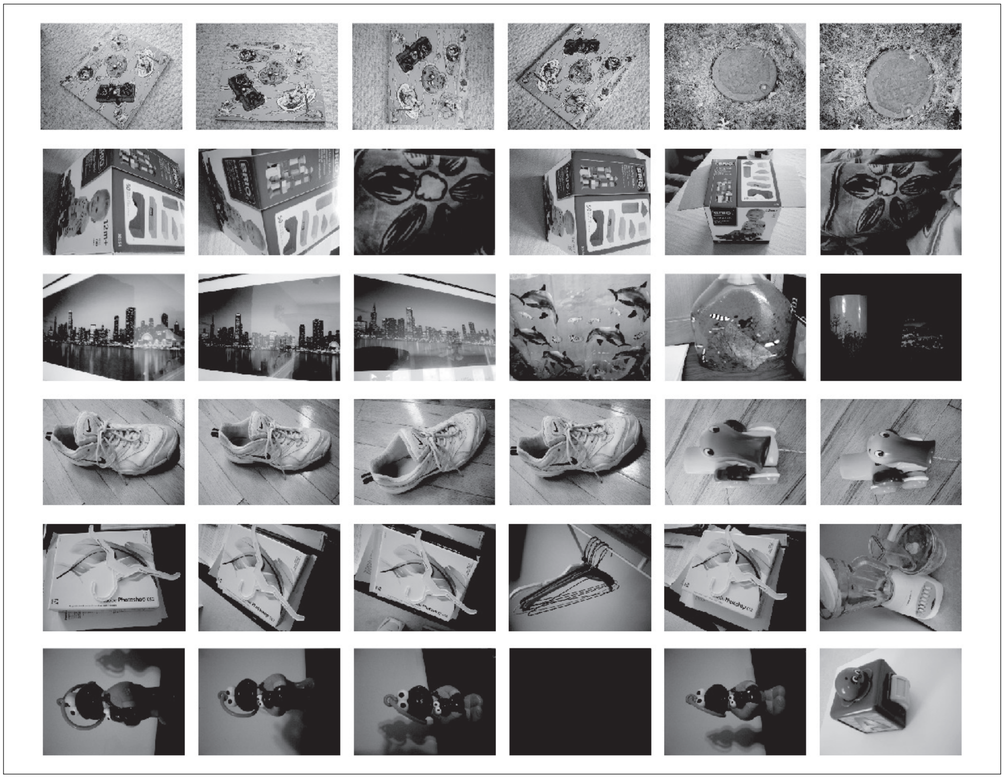


图 7-2：在 ukbench 数据集上用一些查询图像进行搜索给出的一些结果。查询图像在最左边，后面是检索到的前 5 幅图像

利用一些考虑到特征几何关系的准则重排搜索到的靠前结果，可以提高准确率。最常用的方法是在查询图像与靠前图像的特征位置间拟合单应性。

为了提高效率，可以将特征位置存储在数据库中，并由特征的单词 id 决定它们之间的关联（要注意的是，只有在词汇足够大，使单词 id 包含很多准确匹配时，它才起作用）。然而，这需要大幅重写我们上面的数据库和代码，并复杂化表示形式。为了进行说明，我们仅重载靠前图像的特征，并对它们进行匹配。

下面是一个载入所有模型文件并用单应性对靠前的图像进行重排的完整例子：

```
import pickle
import sift
import imagesearch
import homography

# 载入图像列表和词汇
with open('ukbench_imlist.pkl','rb') as f:
```

```
  imlist = pickle.load(f)
  featlist = pickle.load(f)

nbr_images = len(imlist)

with open('vocabulary.pkl', 'rb') as f:
  voc = pickle.load(f)


src = imagesearch.Searcher('test.db',voc)

# 查询图像的索引号和返回的搜索结果数目
q_ind = 50
nbr_results = 20

# 常规查询
res_reg = [w[1] for w in src.query(imlist[q_ind])[:nbr_results]]
print 'top matches (regular):', res_reg

# 载入查询图像特征
q_locs,q_descr = sift.read_features_from_file(featlist[q_ind])
fp = homography.make_homog(q_locs[:,:2].T)

# 用 RANSAC 模型拟合单应性
model = homography.RansacModel()

rank = {}
# 载入搜索结果的图像特征
for ndx in res_reg[1:]:
  locs,descr = sift.read_features_from_file(featlist[ndx])

  # 获取匹配数
  matches = sift.match(q_descr,descr)
  ind = matches.nonzero()[0]
  ind2 = matches[ind]
  tp = homography.make_homog(locs[:,:2].T)

  # 计算单应性，对内点计数。如果没有足够的匹配数则返回空列表
  try:
    H,inliers = homography.H_from_ransac(fp[:,ind],tp[:,ind2],model,match_theshold=4)
  except:
    inliers = []

  # 存储内点数
  rank[ndx] = len(inliers)
```

```
# 将字典排序，以首先获取最内层的内点数
sorted_rank = sorted(rank.items(), key=lambda t: t[1], reverse=True)
res_geom = [res_reg[0]]+[s[0] for s in sorted_rank]
print 'top matches (homography):', res_geom

# 显示靠前的搜索结果
imagesearch.plot_results(src,res_reg[:8])
imagesearch.plot_results(src,res_geom[:8])
```

首先，载入图像列表、特征列表（分别包含图像文件名和 SIFT 特征文件）及词汇。然后，创建一个 Searcher 对象，执行定期查询，并将结果保存在 res_reg 列表中。然后载入 res_reg 列表中每一幅图像的特征，并和查询图像进行匹配。单应性通过计算匹配数和计数内点数得到。最终，我们可以通过减少内点的数目对包含图像索引和内点数的字典进行排序。打印搜索结果列表到控制台，并可视化检索靠前的图像。

输出结果如下：

```
top matches (regular): [39, 22, 74, 82, 50, 37, 38, 17, 29, 68, 52, 91, 15, 90, 31, ... ]
top matches (homography): [39, 38, 37, 45, 67, 68, 74, 82, 15, 17, 50, 52, 85, 22, 87, ... ]
```

图 7-3 给出了常规查询和对常规查询重新排序后的一些样例结果。

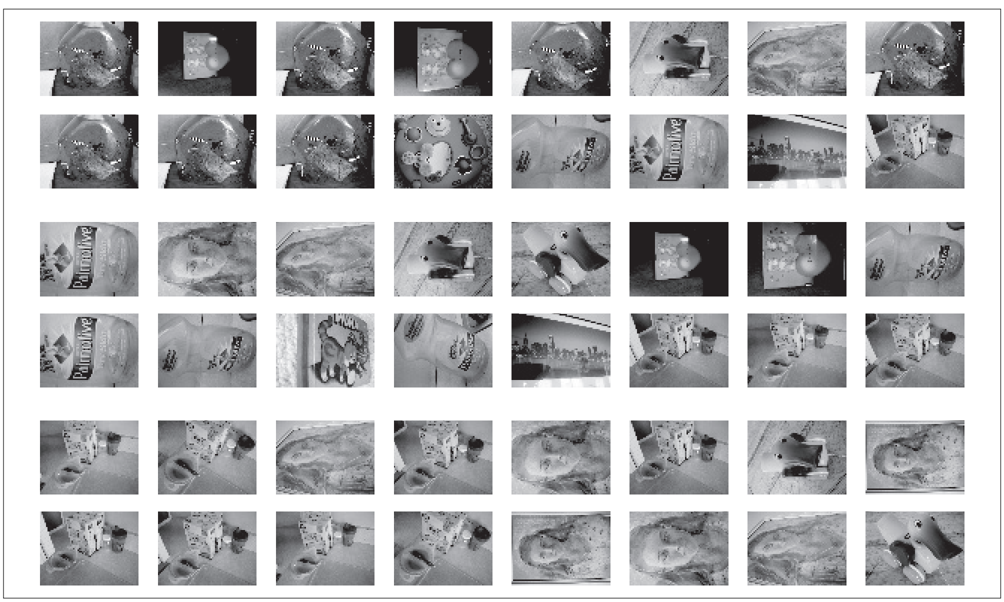


图 7-3：基于几何一致性用单应性对搜索结果进行重排后一些实例搜索结果。在每一个例子中，上一行是没有常规查询的结果，下一行是重排后的结果

7.6 建立演示程序及Web应用

作为本章关于图像搜索的最后一节，我们看一个用 Python 建立演示程序和 Web 应用的简单方法。通过将演示程序变成 Web 页，你便自动获得了跨平台支持，并以最低环境配置需求展示、分享你项目。我们在本节会完整给出一个创建简单图像搜索引擎的示例。

7.6.1 用CherryPy创建Web应用

为了建立这些演示程序，我们将采用 CherryPy 包，参见 http://www.cherrypy.org。CherryPy 是一个纯 Python 轻量级 Web 服务器，使用面向对象模型。CherryPy 的安装和配置细节参见附录 A。这里假设你已经学习了 CherryPy 实例教程，并对 CherryPy 的工作方式有了初步的了解，我们可以以本章创建的图像 Searcher 类为基础，来创建一个图像搜索 Web 演示程序。

7.6.2 图像搜索演示程序

首先，我们需要用一些 HTML 标签进行初始化，并用 Pickle 载入数据。另外，还需要有与数据库进行交互的 Searcher 对象词汇。创建一个名为 searchdemo.py 的文件，并添加下面具有两个方法的 Search Demo 类：

```
import cherrypy, os, urllib, pickle
import imagesearch

class SearchDemo(object):

  def __init__(self):
    # 载入图像列表
    with open('webimlist.txt') as f:
      self.imlist = f.readlines()

  self.nbr_images = len(self.imlist)
  self.ndx = range(self.nbr_images)

  # 载入词汇
  with open('vocabulary.pkl', 'rb') as f:
    self.voc = pickle.load(f)

  # 设置可以显示多少幅图像
  self.maxres = 15

  # html 的头部和尾部
```

```
    self.header = """
      <!doctype html>
      <head>
      <title>Image search example</title>
      </head>
      <body>
      """
    self.footer = """
      </body>
      </html>
      """

  def index(self,query=None):
    self.src = imagesearch.Searcher('web.db',self.voc)

    html = self.header
    html += """
      <br />
      Click an image to search. <a href='?query='>Random selection</a> of images.
      <br /><br />
      """
    if query:
      # 查询数据库并获取靠前的图像
      res = self.src.query(query)[:self.maxres]
      for dist,ndx in res:
        imname = self.src.get_filename(ndx)
        html += "<a href='?query="+imname+"'>"
        html += "<img src='"+imname+"' width='100' />"
        html += "</a>"
    else:
      # 如果没有查询图像，则显示随机选择的图像
      random.shuffle(self.ndx)
      for i in self.ndx[:self.maxres]:
        imname = self.imlist[i]
        html += "<a href='?query="+imname+"'>"
        html += "<img src='"+imname+"' width='100' />"
        html += "</a>"

      html += self.footer
      return html

    index.exposed = True

cherrypy.quickstart(SearchDemo(), '/',
          config=os.path.join(os.path.dirname(__file__), 'service.conf'))
```

你可以看到，这个简单的演示程序包含了单个类，该类包含一个初始化 __int__() 方

法和一个“索引”页面 index() 方法（本例中只有一个页面）。这两个方法可以自动地映射至 URL，并且方法中的参数可以直接传递到 URL 中。index 方法里有一个查询参数，在本例中，该参数是查询图像，用来对其他图像排序。如果该参数是空的，就会随机显示一些图像。

```
index.exposed = True
```

这一行使索引 URL 可以被访问，上面 searchsemo.py 中紧接着该行的最后一行通过读取 service.conf 配置文件开启 CherryPy Web 服务器。在这个例子中，我们的配置文件如下：

```
[global]
server.socket_host = "127.0.0.1"
server.socket_port = 8080
server.thread_pool = 50
tools.sessions.on = True

[/]
tools.staticdir.root = "tmp/"
tools.staticdir.on = True
tools.staticdir.dir = ""
```

第一部分指定使用的 IP 地址和端口，第二部分确保本地文件夹可以读取（本例中文件夹为 tmp/），注意文件夹下存放的是你的图像库。

> 如果你打算将它展示给别人看，不要在这个文件夹下存放任何秘密的东西，因为文件夹下所有的内容都可以通过 CherryPy 访问。

从命令行开启你的 Web 服务器：

```
$ python searchdemo.py
```

打开浏览器，在地址栏输入 http://127.0.0.1:8080/，你可以看到随机挑选出来的图像的初始页面，类似于图 7-4 中的上图所示。点击一幅图像进行查询，会显示出搜索出来的前几幅图像，在搜索出来的图像中单击某图像可以开始新的查询。此外，页面上有一个链接，点击后可以返回原来随机选择的状态（通过一个空查询）。图 7-4 是一些查询示例。

该例子完整展示了从 Web 页面到数据库查询以及结果显示整个综合过程。当然，这只是一个基本的原型，并且你可以在该基础上进行改进，例如添加样式表使它更漂亮，或使它能够上传图像进行查询。

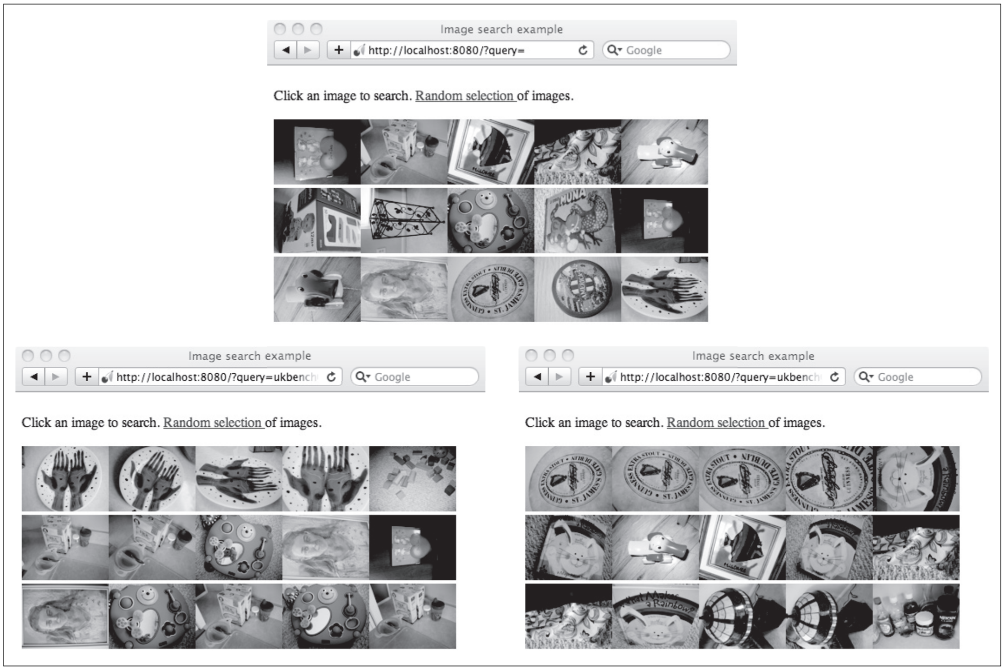


图 7-4：在 ukbench 数据集上进行搜索的示例。上方是开始页面，显示了一些随机选择的图像；下方是一些查询示例。左上角是查询图像，之后的是搜索到的一些结果靠前的图像

练习

(1) 尝试只用查询图像中的部分单词构建候选图像列表来加速查询，用 idf 权重作为准则来挑选单词。

(2) 在你的词汇中，比如前 10%，实现一个最常用的视觉单词停用词列表，在搜索的时候忽略这些单词，看看搜索结果有何改善？

(3) 通过保存所有映射到一个给定单词 id 的所有图像特征，对视觉单词进行可视化。在给定的尺度下，在特征位置环绕处剪切图像块，并将它们画在同一图形窗口中。对于给定的单词，这些图像块看起来一样吗？

(4) 在 `query()` 方法中，用不同的距离度量及加权进行实验，用 `compute_ukbench_score()` 计算得出的分数度量你的改进是否有效。

(5) 在整个章节中，我们的词汇仅用到了 SIFT 特征，正如你在图 7-2 中看到的结果，它完全抛弃了颜色信息。试着添加颜色描述符看你是否能够改进搜索的结果。

(6) 对于大量词汇，用数组来表示视觉单词词频效率很低，原因在于数组中很多元素都是 0（比如考虑这样一种情形，有数十万单词，而图像却只有 1000 个特征）。

一种克服这种低效的办法是将字典作为稀疏数组表示，用自定义一个稀疏类替代这些数组，并在自定义的稀疏类中添加一些必要的方法。或作为另一种选择，你可以采用 `scipy.sparse` 模块。

(7) 你如果增加词汇的大小，聚类时间也会相应增长，并且特征投影到单词的过程更加缓慢。用层次 K-means 聚类算法实现一个层次词汇，看它是怎样提高可伸缩性的，详情参阅文献 [23]。

第 8 章

图像内容分类

本章介绍图像分类和图像内容分类算法。首先，我们介绍一些简单而有效的方法和目前一些性能最好的分类器，并运用它们解决两类和多类分类问题，最后展示两个用于手势识别和目标识别的应用实例。

8.1 K邻近分类法（KNN）

在分类方法中，最简单且用得最多的一种方法之一就是 KNN（K-Nearest Neighbor ,K 邻近分类法），这种算法把要分类的对象（例如一个特征向量）与训练集中已知类标记的所有对象进行对比，并由 *k* 近邻对指派到哪个类进行投票。这种方法通常分类效果较好，但是也有很多弊端：与 K-means 聚类算法一样，需要预先设定 *k* 值，*k* 值的选择会影响分类的性能；此外，这种方法要求将整个训练集存储起来，如果训练集非常大，搜索起来就非常慢。对于大训练集，采取某些装箱形式通常会减少对比的次数[1] 从积极的一面来看，这种方法在采用何种距离度量方面是没有限制的；实际上，对于你所能想到的东西它都可以奏效，但这并不意味着对任何东西它的分类性能都很好。另外，这种算法的可并行性也很一般。

实现最基本的 KNN 形式非常简单。给定训练样本集和对应的标记列表，下面的代码可以用来完成这一工作。这些训练样本和标记可以在一个数组里成行摆放或者干脆摆放列表里，训练样本可能是数字、字符串等任何你喜欢的形状。将定义的类对象添加到名为 knn.py 的文件里：

注 1：另一个选择是只保留从训练集挑选出的子集，但这会影响分类准确率。

```
class KnnClassifier(object):

  def __init__(self,labels,samples):
    """ 使用训练数据初始化分类器 """

    self.labels = labels
    self.samples = samples

  def classify(self,point,k=3):
    """ 在训练数据上采用 k 近邻分类，并返回标记 """

    # 计算所有训练数据点的距离
    dist = array([L2dist(point,s) for s in self.samples])

    # 对它们进行排序
    ndx = dist.argsort()

    # 用字典存储 k 近邻
    votes = {}
    for i in range(k):
      label = self.labels[ndx[i]]
      votes.setdefault(label,0)
      votes[label] += 1

    return max(votes)

def L2dist(p1,p2):
  return sqrt( sum( (p1-p2)**2) )
```

定义一个类并用训练数据初始化非常简单；每次想对某些东西进行分类时，用 KNN 方法，我们就没有必要存储并将训练数据作为参数来传递。用一个字典来存储邻近标记，我们便可以用文本字符串或数字来表示标记。在这个例子中，我们用欧式距离 (L_2) 进行度量，当然，如果你有其他的度量方式，只需要将其作为函数添加到上面代码的最后。

8.1.1　一个简单的二维示例

我们首先建立一些简单的二维示例数据集来说明并可视化分类器的工作原理，下面的脚本将创建两个不同的二维点集，每个点集有两类，用 Pickle 模块来保存创建的数据：

```
from numpy.random import randn
import pickle

# 创建二维样本数据
n = 200
```

```
# 两个正态分布数据集
class_1 = 0.6 * randn(n,2)
class_2 = 1.2 * randn(n,2) + array([5,1])
labels = hstack((ones(n),-ones(n)))

# 用 Pickle 模块保存
with open('points_normal.pkl', 'w') as f:
  pickle.dump(class_1,f)
  pickle.dump(class_2,f)
  pickle.dump(labels,f)


# 正态分布，并使数据成环绕状分布
class_1 = 0.6 * randn(n,2)
r = 0.8 * randn(n,1) + 5
angle = 2*pi * randn(n,1)
class_2 = hstack((r*cos(angle),r*sin(angle)))
labels = hstack((ones(n),-ones(n)))

# 用 Pickle 保存
with open('points_ring.pkl', 'w') as f:
  pickle.dump(class_1,f)
  pickle.dump(class_2,f)
  pickle.dump(labels,f)
```

用不同的保存文件名运行该脚本两次，例如第一次用代码中的文件名进行保存，第二次将代码中的 points_normal_t.pkl 和 points_ring_pkl 分别改为 points_normal_test.pkl 和 points_ring_test.pkl 进行保存。你将得到 4 个二维数据集文件，每个分布都有两个文件，我们可以将一个用来训练，另一个用来做测试。

让我们看看怎么用 KNN 分类器来完成，用下面的代码来创建一个脚本：

```
import pickle
import knn
import imtools

# 用 Pickle 载入二维数据点
with open('points_normal.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

model = knn.KnnClassifier(labels,vstack((class_1,class_2)))
```

这里用 Pickle 模块来创建一个 kNN 分类器模型。现在在上面代码后添加下面的代码：

```
# 用 Pickle 模块载入测试数据
with open('points_normal_test.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# 在测试数据集的第一个数据点上进行测试
print model.classify(class_1[0])
```

上面代码载入另一个数据集（测试数据集），并在你的控制台上打印第一个数据点估计出来的类标记。

为了可视化所有测试数据点的分类，并展示分类器将两个不同的类分开得怎样，我们可以添加这些代码：

```
# 定义绘图函数
def classify(x,y,model=model):
  return array([model.classify([xx,yy]) for (xx,yy) in zip(x,y)])

# 绘制分类边界
imtools.plot_2D_boundary([-6,6,-6,6],[class_1,class_2],classify,[1,-1])
show()
```

这里我们创建了一个简短的辅助函数以获取 x 和 y 二维坐标数组和分类器，并返回一个预测的类标记数组。现在我们把函数作为参数传递给实际的绘图函数。把下面的函数添加到文件 imtools 中：

```
def plot_2D_boundary(plot_range,points,decisionfcn,labels,values=[0]):
  """Plot_range 为（xmin，xmax，ymin，ymax），points 是类数据点列表，
    decisionfcn 是评估函数，labels 是函数 decidionfcn 关于每个类返回的标记列表 """

  clist = ['b','r','g','k','m','y'] # 不同的类用不同的颜色标识

  # 在一个网格上进行评估，并画出决策函数的边界
  x = arange(plot_range[0],plot_range[1],.1)
  y = arange(plot_range[2],plot_range[3],.1)
  xx,yy = meshgrid(x,y)
  xxx,yyy = xx.flatten(),yy.flatten() # 网格中的 x，y 坐标点列表
  zz = array(decisionfcn(xxx,yyy))
  zz = zz.reshape(xx.shape)
  # 以 values 画出边界
  contour(xx,yy,zz,values)

  # 对于每类，用 * 画出分类正确的点，用 o 画出分类不正确的点
  for i in range(len(points)):
```

```
        d = decisionfcn(points[i][:,0],points[i][:,1])
        correct_ndx = labels[i]==d
        incorrect_ndx = labels[i]!=d
        plot(points[i][correct_ndx,0],points[i][correct_ndx,1],'*',color=clist[i])
        plot(points[i][incorrect_ndx,0],points[i][incorrect_ndx,1],'o',color=clist[i])

    axis('equal')
```

这个函数需要一个决策函数（分类器），并且用 meshgrid() 函数在一个网格上进行预测。决策函数的等值线可以显示边界的位置，默认边界为零等值线。画出来的结果如图 8-1 所示，正如你所看到的，kNN 决策边界适用于没有任何明确模型的类分布。

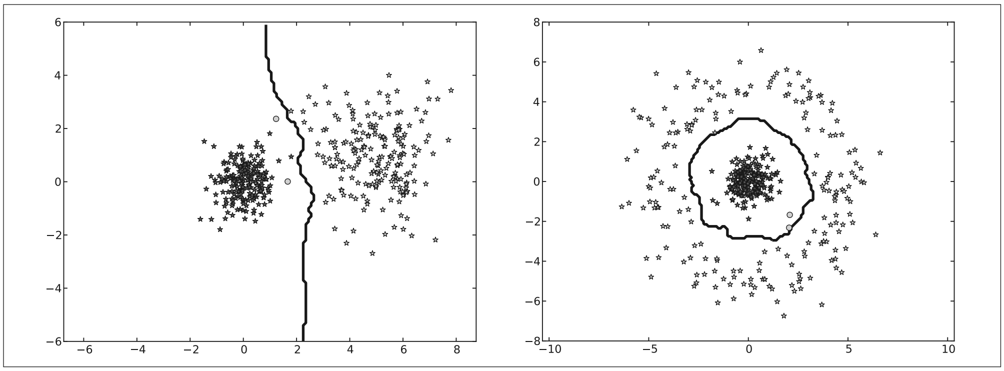


图 8-1：用 K 邻近分类器分类二维数据。每个示例中，不同颜色代表类标记，正确分类的点用星号表示，分类错误的点用圆点表示，曲线是分类器的决策边界

8.1.2 用稠密SIFT作为图像特征

我们来看如何对图像进行分类。要对图像进行分类，我们需要一个特征向量来表示一幅图像。在聚类一章我们用平均 RGB 像素值和 PCA 系数作为图像的特征向量；这里我们会介绍另外一种表示形式，即稠密 SIFT 特征向量。

在整幅图像上用一个规则的网格应用 SIFT 描述子可以得到稠密 SIFT 的表示形式[1]，我们可以利用 2.2 节的可执行脚本，通过添加一些额外的参数来得到稠密 SIFT 特征。创建一个名为 dsift.py 的文件，并添加下面代码到该文件中：

```
import sift

def process_image_dsift(imagename,resultname,size=20,steps=10,
        force_orientation=False,resize=None):
```

注 1：另一个常见的名字是方向梯度直方图（HOG）。

```
""" 用密集采样的 SIFT 描述子处理一幅图像，并将结果保存在一个文件中。可选的输入：
  特征的大小 size，位置之间的步长 steps，是否强迫计算描述子的方位 force_orientation
  （False 表示所有的方位都是朝上的），用于调整图像大小的元组 """

im = Image.open(imagename).convert('L')
if resize!=None:
  im = im.resize(resize)
m,n = im.size

if imagename[-3:] != 'pgm':
  # 创建一个 pgm 文件
  im.save('tmp.pgm')
  imagename = 'tmp.pgm'

# 创建帧，并保存到临时文件
scale = size/3.0
x,y = meshgrid(range(steps,m,steps),range(steps,n,steps))
xx,yy = x.flatten(),y.flatten()
frame = array([xx,yy,scale*ones(xx.shape[0]),zeros(xx.shape[0])])
savetxt('tmp.frame',frame.T,fmt='%03.3f')

if force_orientation:
  cmmd = str("sift "+imagename+" --output="+resultname+
        " --read-frames=tmp.frame --orientations")
else:
  cmmd = str("sift "+imagename+" --output="+resultname+
        " --read-frames=tmp.frame")
os.system(cmmd)
print 'processed', imagename, 'to', resultname
```

对比 2.2 节的 process_image() 函数，为了使用命令行处理，我们用 savetxt() 函数将帧数组存储在一个文本文件中，该函数的最后一个参数可以在提取描述子之前对图像的大小进行调整，例如，传递参数 imsize=(100, 100) 会将图像调整为 100 × 100 像素的方形图像。最后，如果 force_orientation 为真，则提取出来的描述子会基于局部主梯度方向进行归一化；否则，则所有的描述子的方向只是简单地朝上。

利用类似下面的代码可以计算稠密 SIFT 描述子，并可视化它们的位置：

```
import dsift,sift

dsift.process_image_dsift('empire.jpg','empire.sift',90,40,True)
l,d = sift.read_features_from_file('empire.sift')

im = array(Image.open('empire.jpg'))
sift.plot_features(im,l,True)
show()
```

使用用于定位描述子的局部梯度方向（`force_orientation` 设置为真），该代码可以在整个图像中计算出稠密 SIFT 特征。图 8-2 显示出了这些位置。

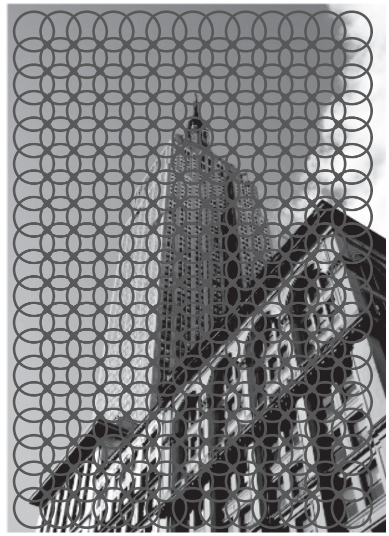


图 8-2：在一幅图像上应用稠密 SIFT 描述子的例子

8.1.3 图像分类：手势识别

在这个应用中，我们会用稠密 SIFT 描述子来表示这些手势图像，并建立一个简单的手势识别系统。我们用静态手势（Static Hand Posture）数据库（参见 http://www.idiap.ch/resource/gestures/）中的一些图像进行演示。在该数据库主页上下载数据较小的测试集 test set 16.3Mb，将下载后的所有图像放在一个名为 uniform 的文件夹里，每一类均分两组，并分别放入名为 train 和 test 的两个文件夹中。

用上面的稠密 SIFT 函数对图像进行处理，可以得到所有图像的特征向量。这里，再次假设列表 `imlist` 中包含了所有图像的文件名，可以通过下面的代码得到每幅图像的稠密 SIFT 特征：

```
import dsift

# 将图像尺寸调为 (50,50)，然后进行处理
for filename in imlist:
  featfile = filename[:-3]+'dsift'
  dsift.process_image_dsift(filename,featfile,10,5,resize=(50,50))
```

上面代码会对每一幅图像创建一个特征文件，文件名后缀为 .dsift。注意，这里将图像分辨率调成了常见的固定大小。这是非常重要的，否则这些图像会有不同数量的

描述子，从而每幅图像的特征向量长度也不一样，这将导致在后面比较它们时出错。图 8-3 绘制出了一些带有描述子的图像。

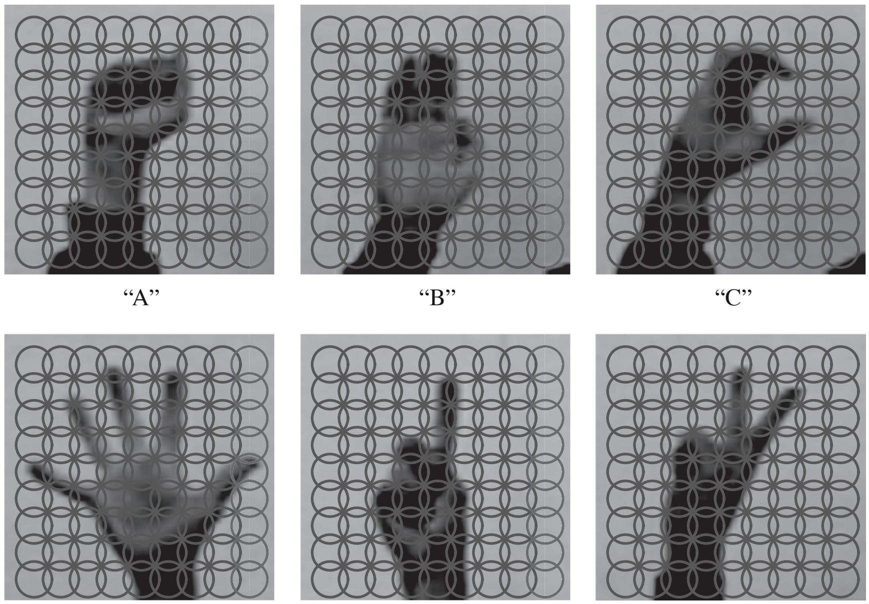


图 8-3：6 类简单手势图像的稠密 SIFT 描述子，图像来源于静态手势（Static Hand Posture）数据库

定义一个辅助函数，用于从文件中读取稠密 SIFT 描述子，如下：

```
import os, sift

def read_gesture_features_labels(path):
  # 对所有以 .dsift 为后缀的文件创建一个列表
  featlist = [os.path.join(path,f) for f in os.listdir(path) if f.endswith('.dsift')]
  # 读取特征
  features = []
  for featfile in featlist:
    l,d = sift.read_features_from_file(featfile)
    features.append(d.flatten())
  features = array(features)

# 创建标记
labels = [featfile.split('/')[-1][0] for featfile in featlist]

return features,array(labels)
```

然后，我们可以用下面的脚本读取训练集、测试集的特征和标记信息：

```
features,labels = read_gesture_features_labels('train/')

test_features,test_labels = read_gesture_features_labels('test/')

classnames = unique(labels)
```

这里，我们用文件名的第一个字母作为类标记，用 NumPy 的 unique() 函数可以得到一个排序后唯一的类名称列表。

现在我们可以在该数据上使用前面的 K 近邻代码：

```
# 测试 KNN
k = 1
knn_classifier = knn.KnnClassifier(labels,features)
res = array([knn_classifier.classify(test_features[i],k) for i in
         range(len(test_labels))])

# 准确率
acc = sum(1.0*(res==test_labels)) / len(test_labels)
print 'Accuracy:', acc
```

首先，用训练数据及其标记作为输入，创建分类器对象；然后，我们在整个测试集上遍历并用 classify() 方法对每幅图像进行分类。将布尔数组和 1 相乘并求和，可以计算出分类的正确率。由于该例中真值为 1，所以很容易计算出正确分类数。它应该会打印出一个类似下面的结果：

```
Accuracy: 0.811518324607
```

这说明该例中有 81% 的图像是正确的。该结果会随 k 值及稠密 SIFT 图像描述子参数的选择而变化。

虽然上面的正确率显示了对于一给定的测试集有多少图像是正确分类的，但是它并没有告诉我们哪些手势难以分类，或会犯哪些典型错误。混淆矩阵是一个可以显示每类有多少个样本被分在每一类中的矩阵，它可以显示错误的分布情况，以及哪些类是经常相互“混淆”的。

下面的函数会打印出标记及相应的混淆矩阵：

```
def print_confusion(res,labels,classnames):

  n = len(classnames)
```

```
# 混淆矩阵
class_ind = dict([(classnames[i],i) for i in range(n)])

confuse = zeros((n,n))
for i in range(len(test_labels)):
  confuse[class_ind[res[i]],class_ind[test_labels[i]]] += 1

print 'Confusion matrix for'
print classnames
print confuse
```

运行：

```
print_confusion(res,test_labels,classnames)
```

打印输出应该如下：

```
Confusion matrix for
['A' 'B' 'C' 'F' 'P' 'V']
[[ 26.   0.   2.   0.   1.   1.]
 [  0.  26.   0.   1.   1.   1.]
 [  0.   0.  25.   0.   0.   1.]
 [  0.   3.   0.  37.   0.   0.]
 [  0.   1.   2.   0.  17.   1.]
 [  3.   1.   3.   0.  14.  24.]]
```

上面混淆矩阵显示，本例子中 P（Point）经常被误分为“V”。

8.2　贝叶斯分类器

另一个简单却有效的分类器是*贝叶斯分类器*[1]（或称*朴素贝叶斯分类器*）。贝叶斯分类器是一种基于贝叶斯条件概率定理的概率分类器，它假设特征是彼此独立不相关的（这就是它“朴素”的部分）。贝叶斯分类器可以非常有效地被训练出来，原因在于每一个特征模型都是独立选取的。尽管它们的假设非常简单，但是贝叶斯分类器已经在实际应用中获得显著成效，尤其是对垃圾邮件的过滤。贝叶斯分类器的另一个好处是，一旦学习了这个模型，就没有必要存储训练数据了，只需存储模型的参数。

该分类器是通过将各个特征的条件概率相乘得到一个类的总概率，然后选取概率最高的那个类构造出来的。

注 1：贝叶斯分类器是以 18 世纪英国教学家、牧师托马斯·贝叶斯命名的。

首先让我们看一个使用高斯概率分布模型的贝叶斯分类器基本实现，也就是用从训练数据集计算得到的特征均值和方差来对每个特征单独建模。把下面的 BayesClassifier 类添加到文件 bayes.py 中：

```
class BayesClassifier(object):

  def __init__(self):
    """ 使用训练数据初始化分类器 """

    self.labels = [] # 类标签
    self.mean = [] # 类均值
    self.var = [] # 类方差
    self.n = 0 # 类别数

  def train(self,data,labels=None):
    """ 在数据 data（n×dim 的数组列表）上训练，标记 labels 是可选的，默认为 0…n-1 """

    if labels==None:
      labels = range(len(data))
    self.labels = labels
    self.n = len(labels)

    for c in data:
      self.mean.append(mean(c,axis=0))
      self.var.append(var(c,axis=0))

  def classify(self,points):
    """ 通过计算得出的每一类的概率对数据点进行分类，并返回最可能的标记 """

    # 计算每一类的概率
    est_prob = array([gauss(m,v,points) for m,v in zip(self.mean,self.var)])

    # 获取具有最高概率的索引，该索引会给出类标签
    ndx = est_prob.argmax(axis=0)
    est_labels = array([self.labels[n] for n in ndx])

    return est_labels, est_prob
```

该模型每一类都有两个变量，即类均值和协方差。`train()` 方法获取特征数组列表（每个类对应一个特征数组），并计算每个特征数组的均值和协方差。`classify()` 方法计算数据点构成的数组的类概率，并选概率最高的那个类，最终返回预测的类标记及概率值，同时需要一个高斯辅助函数：

```
def gauss(m,v,x):
  """ 用独立均值 m 和方差 v 评估 d 维高斯分布 """

  if len(x.shape)==1:
    n,d = 1,x.shape[0]
  else:
    n,d = x.shape

  # 协方差矩阵，减去均值
  S = diag(1/v)
  x = x-m
  # 概率的乘积
  y = exp(-0.5*diag(dot(x,dot(S,x.T))))

  # 归一化并返回
  return y * (2*pi)**(-d/2.0) / ( sqrt(prod(v)) + 1e-6)
```

该函数用来计算单个高斯分布的乘积，返回给定一组模型参数 *m* 和 *v* 的概率，更多多元正态分布例子可以参见 http://en.wikipedia.org/wiki/Multivariate_normal_distribution。

将该贝叶斯分类器用于上一节的二维数据，下面的脚本将载入上一节中的二维数据，并训练出一个分类器：

```
import pickle
import bayes
import imtools

# 用 Pickle 模块载入二维样本点
with open('points_normal.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# 训练贝叶斯分类器
bc = bayes.BayesClassifier()
bc.train([class_1,class_2],[1,-1])
```

下面我们可以载入上一节中的二维测试数据对分类器进行测试：

```
# 用 Pickle 模块载入测试数据
with open('points_normal_test.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)
```

```
# 在某些数据点上进行测试
print bc.classify(class_1[:10])[0]

# 绘制这些二维数据点及决策边界
def classify(x,y,bc=bc):
  points = vstack((x,y))
  return bc.classify(points.T)[0]

imtools.plot_2D_boundary([-6,6,-6,6],[class_1,class_2],classify,[1,-1])
show()
```

该脚本会将前 10 个二维数据点的分类结果打印输出到控制台，输出结果如下：

```
[1 1 1 1 1 1 1 1 1 1]
```

我们再次用一个辅助函数 classify() 在一个网格上评估该函数来可视化这一分类结果。两个数据集的分类结果如图 8-4 所示；该例中，决策边界是一个椭圆，类似于二维高斯函数的等值线。

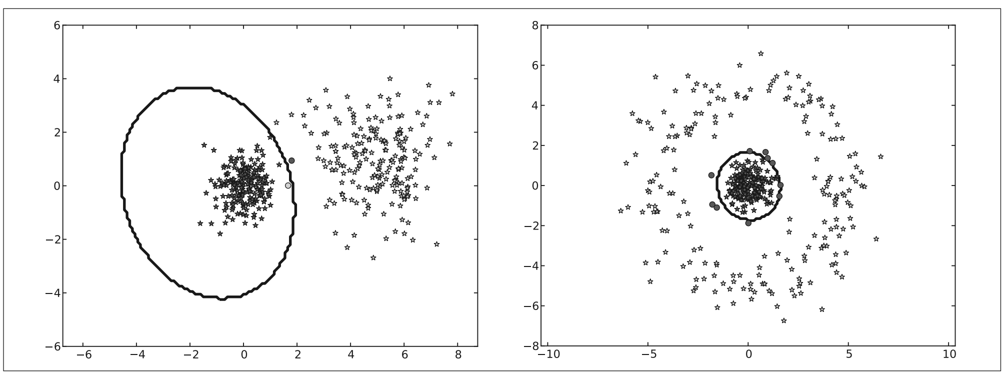


图 8-4：用贝叶斯分类器对二维数据进行分类。每个例子中的颜色代表了类标记。正确分类的点用星号表示，误错分类的点用圆点表示，曲线是分类器的决策边界

用PCA降维

现在，我们尝试手势识别问题。由于稠密 SIFT 描述子的特征向量十分庞大（从前面的例子可以看到，参数的选取超过了 10 000），在用数据拟合模型之前进行降维处理是一个很好的想法。主成分分析，即 PCA（见 1.3 节），非常适合用于降维。下面的脚本就是用 pca.py 中的 PCA 进行降维：

```
import pca

V,S,m = pca.pca(features)

# 保持最重要的成分
V = V[:50]
features = array([dot(V,f-m) for f in features])
test_features = array([dot(V,f-m) for f in test_features])
```

这里的 features 和 test_features 与 K 邻近中的例子中加载的数组是一样的。在本例中，我们在训练数据上用 PCA 降维，并保持在这 50 维具有最大的方差。这可以通过均值 *m*（是在训练数据上计算得到的）并与基向量 V 相乘做到。对测试数据进行同样的转换。

训练并测试贝叶斯分类器如下：

```
# 测试贝叶斯分类器
bc = bayes.BayesClassifier()
blist = [features[where(labels==c)[0]] for c in classnames]

bc.train(blist,classnames)
res = bc.classify(test_features)[0]
```

由于 BayesClassifier 需要获取数组列表（每一类对应一个数组），在把数据传递给 train() 函数之前，我们需要对数据进行转换。因为我们目前还不需要概率，所以只需返回预测的类标记。

检查分类准确率：

```
acc = sum(1.0*(res==test_labels)) / len(test_labels)
print 'Accuracy:', acc
```

输出如下结果：

```
Accuracy: 0.717277486911
```

检查混淆矩阵：

```
print_confusion(res,test_labels,classnames)
```

输出如下结果：

```
Confusion matrix for
['A' 'B' 'C' 'F' 'P' 'V']
[[ 20.  0.  0.  4.  0.  0.]
 [  0. 26.  1.  7.  2.  2.]
 [  1.  0. 27.  5.  1.  0.]
 [  0.  2.  0. 17.  0.  0.]
 [  0.  1.  0.  4. 22.  1.]
 [  8.  2.  4.  1.  8. 25.]]
```

虽然分类效果不如 K 邻近分类器，但是贝叶斯分类器不需要保存任何训练数据，而且只需保存每个类的模型参数。这一结果会随着 PCA 维度选取的不同而发生巨大的变化。

8.3 支持向量机

SVM（Support Vector Machine，支持向量机）是一类强大的分类器，可以在很多分类问题中给出现有水准很高的分类结果。最简单的 SVM 通过在高维空间中寻找一个最优线性分类面，尽可能地将两类数据分开。对于一特征向量 $\boldsymbol{x}$ 的决策函数为：

$$f(\boldsymbol{x})=\boldsymbol{w}\cdot\boldsymbol{x}-b$$

其中 $\boldsymbol{w}$ 是常规的超平面，b 是偏移量常数。该函数月阈值为 0，它能够很好地将两类数据分开，使其一类为正数，另一类为负数。通过在训练集上求解那些带有标记 $y_i\in\{-1,1\}$ 的特征向量 $\boldsymbol{x}_i$ 的最优化问题，使超平面在两类间具有最大分开间隔，从而找到上面决策函数中的参数 $\boldsymbol{w}$ 和 b。该决策函数的常规解是训练集上某些特征向量的线性组合：

$$\boldsymbol{w}=\sum_i \alpha_i y_i \boldsymbol{x}_i$$

所以决策函数可以写为：

$$f(\boldsymbol{x})=\sum_i \alpha_i y_i \boldsymbol{x}_i\cdot\boldsymbol{x}-b$$

这里的 i 是从训练集中选出的部分样本，这里选择的样本称为*支持向量*，因为它们可以帮助定义分类的边界。

SVM 的一个优势是可以使用*核函数*（kernel function）；核函数能够将特征向量映射到另外一个不同维度的空间中，比如高维度空间。通过核函数映射，依然可以保持对决策函数的控制，从而可以有效地解决非线性或者很难的分类问题。用核函数 $K(\boldsymbol{x}_i,\boldsymbol{x})$ 替代上面决策函数中的内积 $\boldsymbol{x}_i\cdot\boldsymbol{x}$。

下面是一些最常见的核函数：

- 线性是最简单的情况，即在特征空间中的超平面是线性的，$K(\boldsymbol{x}_i, \boldsymbol{x})=\boldsymbol{x}_i \cdot \boldsymbol{x}$；
- 多项式用次数为 d 的多项式对特征进行映射，$K(\boldsymbol{x}_i, \boldsymbol{x})=(\gamma\boldsymbol{x}_i \cdot \boldsymbol{x}+r)^d$，$\gamma>0$；
- 径向基函数，通常指数函数是一种极其有效的选择，$K(\boldsymbol{x}_i, \boldsymbol{x})=e^{(-\gamma\|\boldsymbol{x}_i-\boldsymbol{x}\|^2)}$，$\gamma>0$；
- Sigmoid 函数，一个更光滑的超平面替代方案，$K(\boldsymbol{x}_i, \boldsymbol{x})=\tanh(\gamma\boldsymbol{x}_i \cdot \boldsymbol{x}+r)$。

每个核函数的参数都是在训练阶段确定的。

对于多分类问题，通常训练多个 SVM，使每一个 SVM 可以将其中一类与其余类分开，这样的分类器也称为“one-versus-all”分类器。关于 SVM 的更多细节可以参阅文献 [9] 以及在线文档 http://www.support-vector.net/references.html 。

8.3.1 使用LibSVM

LibSVM[7] 是最好的、使用最广泛的 SVM 实现工具包。LibSVM 为 Python 提供了一个良好的接口（也为其他编程语言提供了接口）。关于安装说明，可以参阅附录 A.4。

我们看看 LibSVM 在二维样本数据点上是怎样工作的。下面的脚本会载入在前面 kNN 范例分类中用到的数据点，并用径向基函数训练一个 SVM 分类器：

```
import pickle
from svmutil import *
import imtools

# 用 Pickle 载入二维样本点
with open('points_normal.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# 转换成列表，便于使用 libSVM
class_1 = map(list,class_1)
class_2 = map(list,class_2)
labels = list(labels)
samples = class_1+class_2 # 连接两个列表

# 创建 SVM
prob = svm_problem(labels,samples)
param = svm_parameter('-t 2')

# 在数据上训练 SVM
```

```
m = svm_train(prob,param)

# 在训练数据上分类效果如何？
res = svm_predict(labels,samples,m)
```

我们用与前面一样的方法载入数据集，但是这次需要把数组转成列表，因为 LibSVM 不支持数组对象作为输入。这里，我们用 Python 的内建函数 map() 进行转换，map() 函数中用到了对角一个元素都会进行转换的 list() 函数。紧接着我们创建了一个 svm_problem 对象，并为其设置了一些参数。调用 svm_train() 求解该优化问题用以确定模型参数，然后就可以用该模型进行预测了。最后一行调用 svm_predict()，用求得的模型 *m* 对训练数据分类，并显示出在训练数据中分类的正确率，打印输出结果如下：

```
Accuracy = 100% (400/400) (classification)
```

结果表明该分类器完全分开了训练数据，400 个数据点全部分类正确。

注意，我们在调用方法训练分类器时添加了一个参数选择字符串，这些参数用于控制核函数的类型、等级及其他选项。尽管其中大部分超出了本书范围，但是需要知道两个重要的参数 *t* 和 *k*。参数 *t* 决定了所用核函数的类型，该选项是：

“-*t*”	核函数类型
0	线性函数
1	多项式函数
2	径向基函数（默认）
3	sigmoid 函数

参数 *k* 决定了多项式的次数（默认为 3）。

现在，载入其他数据集，并对该分类器进行测试：

```
# 用 Pickle 模块载入测试数据
with open('points_normal_test.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# 转换成列表，便于使用 LibSVM
class_1 = map(list,class_1)
class_2 = map(list,class_2)

# 定义绘图函数
def predict(x,y,model=m):
  return array(svm_predict([0]*len(x),zip(x,y),model)[0])
```

```
# 绘制分类边界
imtools.plot_2D_boundary([-6,6,-6,6],[array(class_1),array(class_2)],predict,[-1,1])
show()
```

我们需要再次为 LibSVM 将数据转成列表，同时和之前一样，我们定义了一个辅助函数 predict() 来绘制分类的边界。注意，如果无法获取真实的标记，我们用 ([0]*len(x)) 列表来代替标记列表。只要代替的标记列表长度正确，你可以使用任意的列表。图 8-5 显示了两个不同数据集在二维平面上的分布情况。

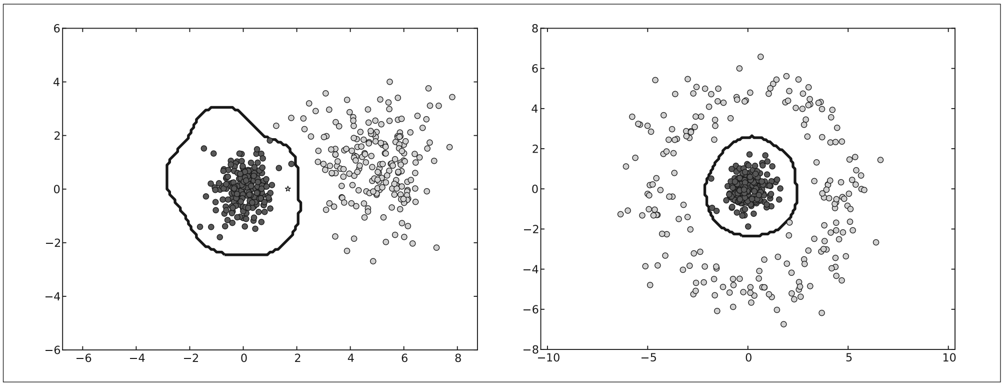


图 8-5：用支持向量机 SVM 对二维数据进行分类。在这两幅图中，我们用不同颜色标识类标记。正确分类的点用星号表示，错误分类的点用圆点表示，曲线是分类器的决策边界

8.3.2 再论手势识别

在多类手势识别问题上使用 LibSVM 相当直观。LibSVM 可以自动处理多个类，我们只需要对数据进行格式化，使输入和输出匹配 LibSVM 的要求。

和之前的例子一样，feature 和 test_features 两个文件中分别数组的形式保存训练数据和测试数据。下面的代码会载入训练数据测试数据，并训练一个线性 SVM 分类器：

```
features = map(list,features)
test_features = map(list,test_features)

# 为标记创建转换函数
transl = {}
for i,c in enumerate(classnames):
  transl[c],transl[i] = i,c

# 创建 SVM
prob = svm_problem(convert_labels(labels,transl),features)
param = svm_parameter('-t 0')
```

```
# 在数据上训练 SVM
m = svm_train(prob,param)

# 在训练数据上分类效果如何
res = svm_predict(convert_labels(labels,transl),features,m)

# 测试 SVM
res = svm_predict(convert_labels(test_labels,transl),test_features,m)[0]
res = convert_labels(res,transl)
```

与之前一样，我们调用 map() 函数将数组转成列表；因为 LibSVM 不能处理字符串标记，所以这些标记也需要转换。字典 transl 会包含一个在字符串和整数标记间的变换。你可以试着在控制台上打印该变换，看看其对应变换关系。参数 -t 0 设置分类器是线性分类器，决策边界在 10 000 维特征原空间中是一个超平面。

现在，对标记进行比较：

```
acc = sum(1.0*(res==test_labels)) / len(test_labels)
print 'Accuracy:', acc

print_confusion(res,test_labels,classnames)
```

用线性核函数得出的分类结果如下：

```
Accuracy: 0.916230366492
Confusion matrix for
['A' 'B' 'C' 'F' 'P' 'V']
[[ 26.  0.  1.  0.  2.  0.]
 [  0. 28.  0.  0.  1.  0.]
 [  0.  0. 29.  0.  0.  0.]
 [  0.  2.  0. 38.  0.  0.]
 [  0.  1.  0.  0. 27.  1.]
 [  3.  0.  2.  0.  3. 27.]]
```

现在，正如我们在 8.2 节所做的，用 PCA 将维数降低到 50，分类正确率变为：

```
Accuracy: 0.890052356021
```

可以看出，当特征向量维数降低到原空间数据维数的 1/200 时，结果并不差（存储支持向量所需占用的空间也减小到原来的 1/200）。

8.4 光学字符识别

作为一个多类问题实例，让我们来理解数独图像。OCR（Optical Character Recognition，

光学字符识别）是一个理解手写或机写文本图像的处理过程。一个常见的例子是通过扫描文件来提取文本，例如书信中的邮政编码或者谷歌图书（http://books.google.com/）里图书馆卷的页数。这里我们看一个简单的在打印的数独图形中识别数字的 OCR 问题。数独是一种数字逻辑游戏，规则是用数字 1~9 填满 9 × 9 的网格，使每一行每一列和每个 3 × 3 的子网格包含数字 1…9[1]。在这个例子中我们只对正确地读取和理解它们感兴趣，对于完成识别后怎样求解这些数独问题我们就留给你。

8.4.1　训练分类器

对于这种分类问题，我们有 10 个类：数字 1…9，以及一些什么也没有的单元格。我们给定什么也没有的单元格的类标号是 0，这样所有类标记就是 0…9。我们会用已经剪切好的数独单元格数据集来训练一个 10 类的分类器 [2] 文件 sudoku_images.zip 中有两个文件夹“ocr data”和“sudokus”，后者包含了不同条件下的数独图像集，我们稍后讲解。现在我们主要来看文件夹“ocr_data”，这个文件夹包含了两个子文件夹，一个装有训练图像，另一个装有测试图像。这些图像文件名的第一个字母是数字（0…9），用以标明它们属于哪类。图 8-6 展示了训练集中的一些样本。图像是灰度图，大概是 80 × 80 像素（有一幅波动）。

8.4.2　选取特征

我们首先要确定选取怎样的特征向量来表示每一个单元格里的图像。有很多不错的选择；这里我们将会用某些简单而有效的特征。输入一个图像，下面的函数将返回一个拉成一组数组后的灰度值特征向量：

```
def compute_feature(im):
  """ 对一个 ocr 图像块返回一个特征向量 """

  # 调整大小，并去除边界
  norm_im = imresize(im,(30,30))
  norm_im = norm_im[3:-3,3:-3]

  return norm_im.flatten()
```

compute_feature() 函数用到 imtools 模块中的尺寸调整函数 imresize()，来减少特征向量的长度。我们还修剪掉了大约 10% 的边界像素，因为这些修剪掉的部分通常是网格线的边缘部分，如图 8-6 所示。

注 1：如果你不熟悉该概念，更多细节参见 http://en.wikipedia.org/ wiki /Sudoku。

注 2：图像来自 Martin Byröd [4], http://www.maths.lth.se/matematikith/personal/byrod，搜集、裁剪于真实的数独照片。

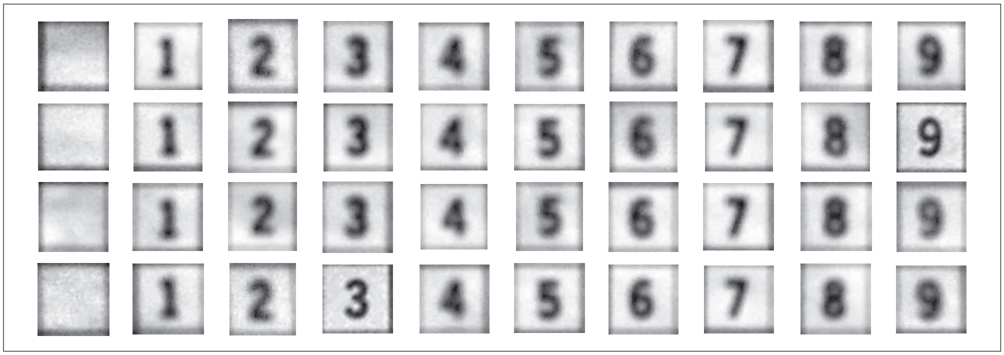


图 8-6：用于训练 10 类数独 OCR 分类器的训练样本图像

现在我们用下面的函数来读取训练数据：

```
def load_ocr_data(path):
""" 返回路径中所有图像的标记及 OCR 特征 """

  # 对以 .jpg 为后缀的所有文件创建一个列表
  imlist = [os.path.join(path,f) for f in os.listdir(path) if f.endswith('.jpg')]
  # 创建标记
  labels = [int(imfile.split('/')[-1][0]) for imfile in imlist]

  # 从图像中创建特征
  features = []
  for imname in imlist:
    im = array(Image.open(imname).convert('L'))
    features.append(compute_feature(im))
return array(features),labels
```

上述代码将每一个 JPEG 文件的文件名中的第一个字母提取出来做类标记，并且这些标记被作为整型数据存储在 lables 列表里；用上面的函数计算出的特征向量存储在一个数组里。

8.4.3 多类支持向量机

在得到了训练数据之后，我们接下来要学习一个分类器，这里将使用多类支持向量机。代码和上一节中的代码类似：

```
from svmutil import *

# 训练数据
features,labels = load_ocr_data('training/')

# 测试数据
test_features,test_labels = load_ocr_data('testing/')
```

```
# 训练一个线性 SVM 分类器
features = map(list,features)
test_features = map(list,test_features)

prob = svm_problem(labels,features)
param = svm_parameter('-t 0')

m = svm_train(prob,param)

# 在训练数据上分类效果如何
res = svm_predict(labels,features,m)

# 在测试集上表现如何
res = svm_predict(test_labels,test_features,m)
```

该代码会训练出一个线性 SVM 分类器，并在测试集上对该分类器的性能进行测试，你可以通过调用最后两个 svm_predict() 函数得到以下输出结果：

```
Accuracy = 100% (1409/1409) (classification)
Accuracy = 99.2979% (990/997) (classification)
```

真是一个极好的结果，训练集中的 1409 张图像在 10 类中都被完美地分准确了，在测试集上识别性能也在 99% 左右。现在我们可以将这一分类器用到经过裁剪的新数独图像上。

8.4.4　提取单元格并识别字符

有了识别单元格内容的分类器后，下一步就是自动地找到这些单元格。一旦我们解决了这个问题，就可以对单元格进行裁剪，并把裁剪后的单元格传给分类器。我们假设数独图像是已经对齐的，其水平和垂直网格线平行于图像的边，如图 8-7 所示。在这些条件下，我们可以对图像进行阈值化处理，并在水平和垂直方向上分别对像素值求和。由于这些经阈值处理的边界值为 1，而其他部分值为 0，所以这些边界处会给出很强的响应，可以告诉我们从何处进行裁剪。

下面函数接受一幅灰度图像和一个方向，返回该方向上的 10 条边界：

```
from scipy.ndimage import measurements

def find_sudoku_edges(im,axis=0):
  """ 对一幅对齐后的数独图像查找单元格的边界 """

# 阈值处理，处理后对每行（列）相加求和
trim = 1*(im<128)
s = trim.sum(axis=axis)

# 寻找连通域
```

```
s_labels,s_nbr = measurements.label(s>(0.5*max(s)))
# 计算各连通域的质心
m = measurements.center_of_mass(s,s_labels,range(1,s_nbr+1))
# 对质心取整，质心即为相线条所在位置
x = [int(x[0]) for x in m]

# 只要检测到 4 条粗线，便在这 4 条粗线之间添加直线
if len(x)==4:
  dx = diff(x)
  x = [x[0],x[0]+dx[0]/3,x[0]+2*dx[0]/3,
      x[1],x[1]+dx[1]/3,x[1]+2*dx[1]/3,
      x[2],x[2]+dx[2]/3,x[2]+2*dx[2]/3,x[3]]

  if len(x)==10:
    return x
  else:
    raise RuntimeError('Edges not detected.')
```

首先对图像进行阈值化处理，对灰度值小于 128 的暗区域赋值为 1，否则为 0；然后在特定的方向上（如 axis=0 或 1）对这些经阈值处理后的像素相加求和。Scipy.ndimage 包含 measurements 模块，该模块在二进制或标记数组中对于计数及测量区域是非常有用的。首先，labels() 找出二进制数组中相连接的部件；该二进制数组是通过求和后取中值并进行阈值化处理得到的。然后，center_of_mass() 函数计算每个独立组件的质心。你可能得到 4 个或 10 个点，这主要依赖于数独平面造型设计（所有的线条是等粗细的或子网格线条比其他的粗）。在 4 个点的情况下，会以一定的间隔插入 6 条直线。如果最后的结果没有 10 条线，则会抛出一个异常。

sudokus 文件夹里包含不同难易程度的数独图像，每幅图像都对应一个包含数独真实值的文件，我们可以用它来检查识别结果。有一些图像已经和图像的边框对齐，从中挑选一幅图像，用以检查图像裁剪及分类的性能：

```
imname = 'sudokus/sudoku18.jpg'
vername = 'sudokus/sudoku18.sud'
im = array(Image.open(imname).convert('L'))

# 查找单元格边界
x = find_sudoku_edges(im,axis=0)
y = find_sudoku_edges(im,axis=1)

# 裁剪单元格并分类
crops = []
for col in range(9):
  for row in range(9):
    crop = im[y[col]:y[col+1],x[row]:x[row+1]]
    crops.append(compute_feature(crop))
```

```
res = svm_predict(loadtxt(vername),map(list,crops),m)[0]
res_im = array(res).reshape(9,9)

print 'Result:'
print res_im
```

找到边界后，从每一个单元格提取出 crops。将裁剪出来的这些单元格传给同一特征提取函数，并将提取出来的特征作为训练数据保存在一个数组中。通过 loadtxt() 读取数独图像的真实标记，用 svm_predict() 函数对这些特征向量进行分类，在控制台上打印出的结果应该为：

```
Accuracy = 100% (81/81) (classification)
Result:
[[ 0. 0. 0. 0. 1. 7. 0. 5. 0.]
 [ 9. 0. 3. 0. 0. 5. 2. 0. 7.]
 [ 0. 0. 0. 0. 0. 0. 4. 0. 0.]
 [ 0. 1. 6. 0. 0. 4. 0. 0. 2.]
 [ 0. 0. 0. 8. 0. 1. 0. 0. 0.]
 [ 8. 0. 0. 5. 0. 0. 6. 4. 0.]
 [ 0. 0. 9. 0. 0. 0. 0. 0. 0.]
 [ 7. 0. 2. 1. 0. 0. 8. 0. 9.]
 [ 0. 5. 0. 2. 3. 0. 0. 0. 0.]]
```

这里使用的只是其中较简单的图像，请尝试对一些别的数独图像进行识别，看看识别准确率如何，以及分类器在哪些地方出现识别错误。

如果你用一个 9×9 的子图绘制这些经裁剪后的单元格，它们应该和图 8-7（右图）类似。

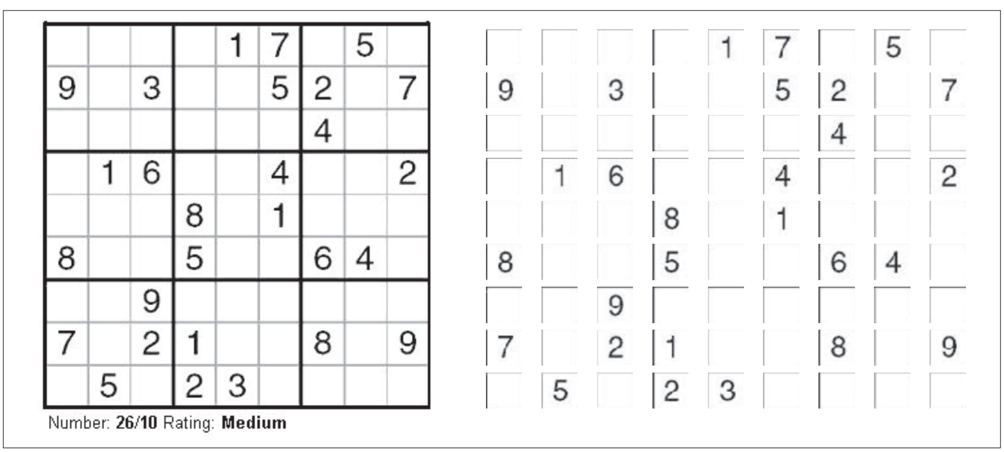


图 8-7：一个检测并裁剪这些数独网格区域的例子：一幅数独网格图像 (左)；9×9 裁剪后的图像，每个独立单元都会被送到 OCR 分类器中 (右)

8.4.5 图像校正

如果你对上面分类器的性能还算满意，那么下一个挑战便是如何将它应用于那些没有对齐的图像。这里我们将用一种简单的图像校正方法来结束本章数独图像识别的例子，使用该校正方法的前提是网格的 4 个角点都已经被检测到或者手工做过标记。图 8-8（左）中是一幅在进行识别时受角度影响剧烈的图像。

一个单应矩阵可以像上面的例子那样映射网格以使边缘能够对齐，我们这里所要做的就是估算该变换矩阵。下面的例子手工标记 4 个角点，然后将图像变换为一个 1000 × 1000 大小的方形图像：

```
from scipy import ndimage
import homography

imname = 'sudoku8.jpg'
im = array(Image.open(imname).convert('L'))

# 标记角点
figure()
imsshow(im)
gray()
x = ginput(4)

# 左上角、右上角、右下角、左下角
fp = array([array([p[1],p[0],1]) for p in x]).T
tp = array([[0,0,1],[0,1000,1],[1000,1000,1],[1000,0,1]]).T

# 估算单应矩阵
H = homography.H_from_points(tp,fp)

# 辅助函数，用于进行几何变换
def warpfcn(x):
  x = array([x[0],x[1],1])
  xt = dot(H,x)
  xt = xt/xt[2]
  return xt[0],xt[1]

# 用全透视变换对图像进行变换
im_g = ndimage.geometric_transform(im,warpfcn,(1000,1000))
```

第 3 章中对很多样本图像进行的仿射变换还达不到对这些数独图像进行校正的要求，这里用到了 scipy.ndinmage 模块中一个更加普遍的变换函数 `geometric_transform()`，该函数获取一个 2D 到 2D 的映射，映射为另一个二维来取代变化矩阵，所以我们需要一个辅助函数（该例中用一个三角形的分段仿射变换），变换后的图像如图 8-8 中右图所示。

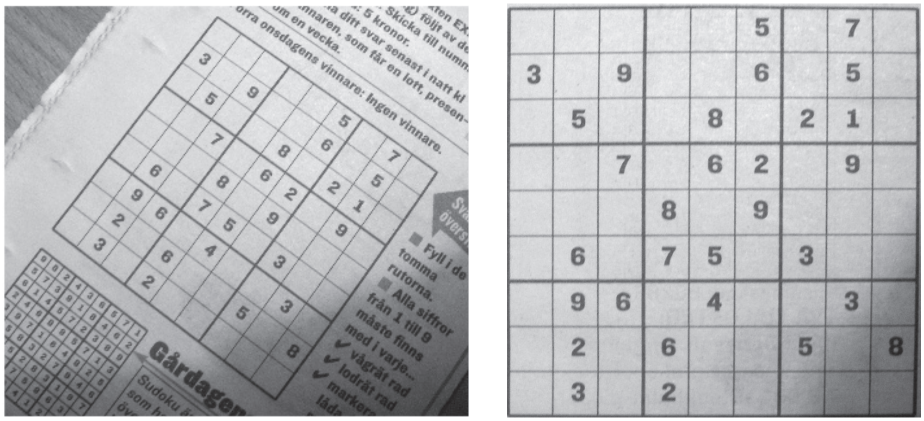


图 8-8：用全透视变换对一幅图像进行校正的例子。四个角点被手工标记的数独原图（左），变换为 1000 × 1000 大小的方形图（右）

这就是整个关于数独图像识别的例子。其中还有很多可以改进的地方，及一些待调查的识别方案。下面的练习中会提到一些，剩下的则留给你自行研究。

练习

(1) KNN 分类器的性能依赖于 *k* 值的选择。试着改变 *k* 值，看分类精度是怎样变化的。画出二维数据集的决策边界，看它们是怎样变化的。

(2) 图 8-3 中的手势数据库文件夹 complex/ 还包含背景更复杂的手势图像。试着在这些图像上训练出一个分类器并对训练出的分类器进行测试。它们在性能上有什么差异？你能够对图像描述子做出一些改进吗？

(3) 对于贝叶斯分类器，在对手势识别特征进行 PCA 降维时，试着改变降维的维数。哪一个维数较好？画出奇异值 *S*，画出的曲线如图 8-9 所示，是一个典型的膝状曲线。在训练数据生成可变性能力和保持较低的维数间，一个较好的折中是在图像变得平缓之前选取一个恰当的维数。

(4) 用一个不同于高斯分布的概率模型修改贝叶斯分类器，例如在训练集上对于每个特征使用频率计数，在不同的数据集上和高斯分布结果进行对比。

(5) 用非线性 SVM 对于手势识别问题进行实验，比如选择多项式核函数，并用 -d 参数逐步增加多项式的阶数，观察在训练集和测试集上分类性能的变化。对于非线性分类器，存在一种潜在的风险，即在某个特定的数据集上，可能在训练集上分类结果很好，但测试集上分类结果很差。这种破坏分类器泛化能力的现象称为过拟合，我们应该避开这种风险。

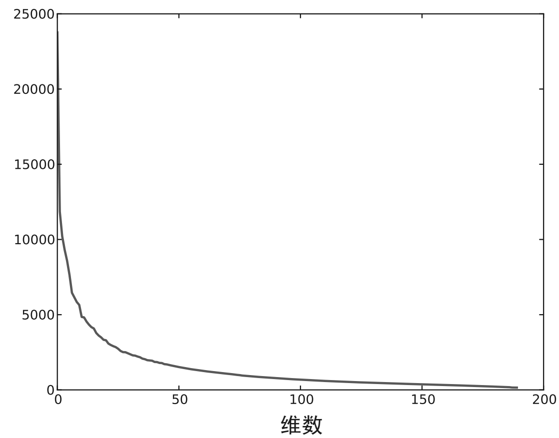


图 8-9：练习 (3) 曲线图

(6) 试着在数独字符识别上用一些更高级的特征向量。如果你需要灵感，参见文献 [4]。

(7) 实现一种能自动对齐数独网格的方法，例如试着用 RANSAC 进行特征检测、进行直线检测，或从 `scipy.ndimage`（http://docs.scipy.org/doc/scipy/reference/ndimage.html）用形态学和测量操作检测这些单元格。另外，作为额外的任务，请试着解决查找方向“向上”的旋转不确定问题。比如，你可以试着旋转校正后的网格，并以 OCR 分类器的分类准确率选出最佳旋转方向。

(8) MNIST 手写数字数据库（http://yann.lecun.com/exdb/mnist）是一个比数独数字图像库更具有挑战性的分类问题。试着提取 MNIST 数据库的一些特征，并用 SVM 对该数据库进行分类。检查你所获得的分类准确率和已知的最佳分类方法（有些方法出奇得好）之间差距。

(9) 如果你想更深入地了解分类器和机器学习算法，可以参阅 `scikit`.learn 包（http://scikit-learn.org/），然后在本章的数据库上尝试其中的一些算法。

第 9 章

图像分割

图像分割是将一幅图像分割成有意义区域的过程。区域可以是图像的前景与背景或图像中一些单独的对象。这些区域可以利用一些诸如颜色、边界或近邻相似性等特征进行构建。本章中，我们将看到一些不同的分割技术。

9.1 图割（Graph Cut）

图论中的图（graph）是由若干节点（有时也称顶点）和连接节点的边构成的集合。图 9-1 给出了一个示例[1]。边可以是有向的（图 9-1 中用箭头示出）或无向的，并且这些可能有与它们相关联的权重。

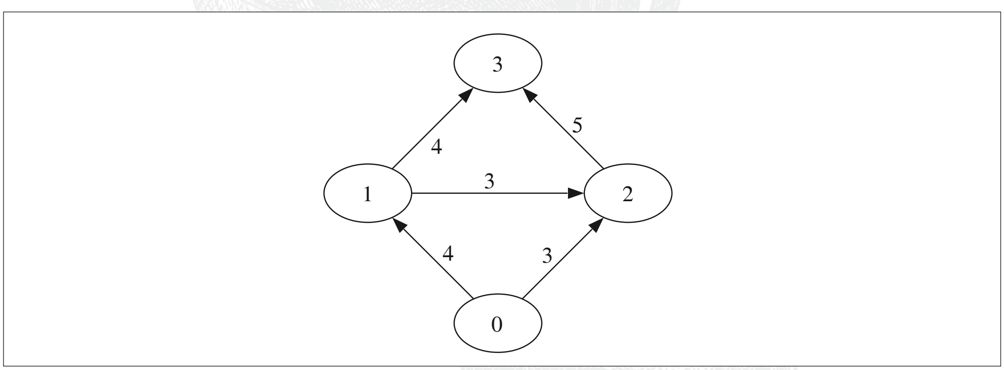


图 9-1：用 python-graph 工具包创建的一个简单有向图

注 1：2.3 节中也出现过图。这次我们要使用图进行图像分割。

图割是将一个有向图分割成两个互不相交的集合，可以用来解决很多计算机视觉方面的问题，诸如立体深度重建、图像拼接和图像分割等计算机视觉方面的不同问题。从图像像素和像素的近邻创建一个图并引入一个能量或“代价”函数，我们有可能利用图割方法将图像分割成两个或多个区域。图割的基本思想是，相似且彼此相近的像素应该划分到同一区域。

图割 C（C 是图中所有边的集合）的“代价”函数定义为所有割的边的权重求合相加：

$$E_{cut} = \sum_{(i,j)\in C} w_{ij} \tag{9.1}$$

w_{ij} 是图中节点 i 到节点 j 的边（i，j）的权重，并且是对割 C 所有的边进行求和。

图割图像分割的思想是用图来表示图像，并对图进行划分以使割代价 E_{cut} 最小。在用图表示图像时，增加两个额外的节点，即源点和汇点；并仅考虑那些将源点和汇点分开的割。

寻找最小割（minimum cut 或 min cut）等同于在源点和汇点间寻找最大流（maximum flow 或 max flow），详见文献 [2]。此外，很多有效的算法都可以解决这些最大流 / 最小割问题。

我们在图割例子中将采用 python-graph 工具包，该工具包包含了许多非常有用的图算法，你可以在 http://code.google.com/p/python-graph/ 下载该工具包并查看文档。随后的例子里，我们要用到 maximum_flow() 函数，该函数用 Edmonds-Karp 算法（http://en.wikipedia.org/wiki/Edmonds-Karp_algorithm）计算最大流 / 最小割。采用一个完全用 Python 写成工具包的好处是安装容易且兼容性良好；不足是速度较慢。不过，对于小尺寸图像，该工具包的性能足以满足我们的需求，对于较大的图像，需要一个更快的实现。

这里给出一个用 python-graph 工具包计算一幅较小的图 [1] 的最大流 / 最小割的简单例子：

```
from pygraph.classes.digraph import digraph
from pygraph.algorithms.minmax import maximum_flow

gr = digraph()
gr.add_nodes([0,1,2,3])

gr.add_edge((0,1), wt=4)
gr.add_edge((1,2), wt=3)
gr.add_edge((2,3), wt=5)
gr.add_edge((0,2), wt=3)
```

注 1：http://en.wikipedia.org/wiki/Max-flow_min-cut_theorem 上有相同的图作为例子。

```
gr.add_edge((1,3), wt=4)

flows,cuts = maximum_flow(gr,0,3)
print 'flow is:', flows
print 'cut is:', cuts
```

首先，创建有 4 个节点的有向图，4 个节点的索引分别 0...3，然后用 add_edge() 增添边并为每条边指定特定的权重。边的权重用来衡量边的最大流容量。以节点 0 为源点、3 为汇点，计算最大流。打印出流和割结果：

```
flow is: {(0, 1): 4, (1, 2): 0, (1, 3): 4, (2, 3): 3, (0, 2): 3}
cut is: {0: 0, 1: 1, 2: 1, 3: 1}
```

上面两个 python 字典包含了流穿过每条边和每个节点的标记：0 是包含图源点的部分，1 是与汇点相连的节点。你可以手工验证这个割确实是最小的。参见图 9-1。

9.1.1　从图像创建图

给定一个邻域结构，我们可以利用图像像素作为节点定义一个图。这里我们将集中讨论最简单的像素四邻域和两个图像区域（前景和背景）情况。一个四邻域（4-neighborhood）指一个像素与其正上方、正下方、左边、右边的像素直接相连[1]。

除了像素节点外，我们还需要两个特定的节点——“源”点和“汇”点，来分别代表图像的前景和背景。我们将利用一个简单的模型将所有像素与源点、汇点连接起来。

下面给出创建这样一个图的步骤：

- 每个像素节点都有一个从源点的传入边；
- 每个像素节点都有一个到汇点的传出边；
- 每个像素节点都有一条传入边和传出边连接到它的近邻。

为确定边的权重，需要一个能够确定这些像素点之间，像素点与源点、汇点之间边的权重（表示那条边的最大流）的分割模型。与前面一样，像素 i 与像素 j 之间的边的权重记为 w_{ij}，源点到像素 i 的权重记为 w_{si}，像素 i 到汇点的权重记为 w_{it}。

让我们先回顾一下 8.2 节中在像素颜色值上用朴素贝叶斯分类器进行分类的知识。假定我们已经在前景和背景像素（从同一图像或从其他的图像）上训练出了一个贝叶斯分类器，我们就可以为前景和背景计算概率 $p_F(I_i)$ 和 $p_B(I_i)$。这里，I_i 是像素 i 的颜色向量。

现在我们可以为边的权重建立如下模型：

注 1：另一个常见的选择是八邻域，它把对角上的像素也连结起来了。

$$w_{si} = \frac{p_F(I_i)}{p_F(I_i) + p_B(I_i)}$$
$$w_{it} = \frac{p_B(I_i)}{p_F(I_i) + p_B(I_i)}$$
$$w_{ij} = \kappa \, e^{-|I_i - I_j|^2/\sigma}$$

利用该模型，可以将每个像素和前景及背景（源点和汇点）连接起来，权重等于上面归一化后的概率。w_{ij} 描述了近邻间像素的相似性，相似像素权重趋近于 κ，不相似的趋近于 0。参数 σ 表征了随着不相似性的增加，指数次幂衰减到 0 的快慢。

创建一个名为 graphcut.py 的文件，将下面从一幅图像创建图的函数写入该文件中：

```
from pygraph.classes.digraph import digraph
from pygraph.algorithms.minmax import maximum_flow

import bayes

def build_bayes_graph(im,labels,sigma=1e2,kappa=2):
  """ 从像素四邻域建立一个图，前景和背景（前景用 1 标记，背景用 -1 标记，
    其他的用 0 标记）由 labels 决定，并用朴素贝叶斯分类器建模 """

  m,n = im.shape[:2]

  # 每行是一个像素的 RGB 向量
  vim = im.reshape((-1,3))

  # 前景和背景（RGB）
  foreground = im[labels==1].reshape((-1,3))
  background = im[labels==-1].reshape((-1,3))
  train_data = [foreground,background]

  # 训练朴素贝叶斯分类器
  bc = bayes.BayesClassifier()
  bc.train(train_data)

  # 获取所有像素的概率
  bc_lables,prob = bc.classify(vim)
  prob_fg = prob[0]
  prob_bg = prob[1]

  # 用 m*n+2 个节点创建图
  gr = digraph()
  gr.add_nodes(range(m*n+2))
```

```
source = m*n # 倒数第二个是源点
sink = m*n+1 # 最后一个节点是汇点

# 归一化
for i in range(vim.shape[0]):
  vim[i] = vim[i] / linalg.norm(vim[i])

# 遍历所有的节点，并添加边
for i in range(m*n):
  # 从源点添加边
  gr.add_edge((source,i), wt=(prob_fg[i]/(prob_fg[i]+prob_bg[i])))

# 向汇点添加边
gr.add_edge((i,sink), wt=(prob_bg[i]/(prob_fg[i]+prob_bg[i])))

# 向相邻节点添加边
if i%n != 0: # 左边存在
    edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i-1])**2)/sigma)
    gr.add_edge((i,i-1), wt=edge_wt)
if (i+1)%n != 0: # 如果右边存在
    edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i+1])**2)/sigma)
    gr.add_edge((i,i+1), wt=edge_wt)
if i//n != 0: 如果上方存在
    edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i-n])**2)/sigma)
    gr.add_edge((i,i-n), wt=edge_wt)
if i//n != m-1: # 如果下方存在
    edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i+n])**2)/sigma)
    gr.add_edge((i,i+n), wt=edge_wt)

return gr
```

这里我们用到了用 1 标记前景训练数据、用 −1 标记背景训练数据的一幅标记图像。基于这种标记，在 RGB 值上可以训练出一个朴素贝叶斯分类器，然后计算每一像素的分类概率，这些计算出的分类概率便是从源点出来和到汇点去的边的权重；由此可以创建一个节点为 $n \times m+2$ 的图。注意源点和汇点的索引；为了简化像素的索引，我们将最后的两个索引作为源点和汇点的索引。

为了在图像上可视化覆盖的标记区域，我们可以利用 contourf() 函数填充图像（这里指带标记图像）等高线间的区域，参数 alpha 用于设置透明度。将下面函数增加到 graphcut.py 中：

```
def show_labeling(im,labels):
  """ 显示图像的前景和背景区域。前景 labels=1, 背景 labels=-1，其他 labels = 0 """
```

```
    imshow(im)
    contour(labels,[-0.5,0.5])
    contourf(labels,[-1,-0.5],colors='b',alpha=0.25)
    contourf(labels,[0.5,1],colors='r',alpha=0.25)
    axis('off')
```

图建立起来后便需要在最优位置对图进行分割。下面这个函数可以计算最小割并将输出结果重新格式化为一个带像素标记的二值图像：

```
def cut_graph(gr,imsize):
  """ 用最大流对图 gr 进行分割，并返回分割结果的二值标记 """

  m,n = imsize
  source = m*n # 倒数第二个节点是源点
  sink = m*n+1 # 倒数第一个是汇点

  # 对图进行分割
  flows,cuts = maximum_flow(gr,source,sink)

  # 将图转为带有标记的图像
  res = zeros(m*n)
  for pos,label in cuts.items()[:-2]: # 不要添加源点 / 汇点
    res[pos] = label

  return res.reshape((m,n))
```

需要再次注意源点和汇点的索引；我们需要将图像的尺寸作为输入去计算这些索引，在返回分割结果之前要对输出结果进行 reshape() 操作。割以字典返回，需要将它复制到分割标记图像中，这通过返回列表（键，值）的 .item() 方法完成。这里我们再一次略过了列表中最后两个元素。

现在让我们看看怎样利用这些函数来分割一幅图像。下面这个例子会读取一幅图像，从图像的两个矩形区域估算出类概率，然后创建一个图：

```
from scipy.misc import imresize
import graphcut

im = array(Image.open('empire.jpg'))
im = imresize(im,0.07,interp='bilinear')
size = im.shape[:2]

# 添加两个矩形训练区域
labels = zeros(size)
labels[3:18,3:18] = -1
```

```
labels[-18:-3,-18:-3] = 1

# 创建图
g = graphcut.build_bayes_graph(im,labels,kappa=1)

# 对图进行分割
res = graphcut.cut_graph(g,size)

figure()
graphcut.show_labeling(im,labels)

figure()
imshow(res)
gray()
axis('off')

show()
```

我们利用了 imresize() 函数使图像小到适合我们的 Python graph 库；在该例中将图像统一缩放到原图像尺寸的 7%。图分割后将结果和训练区域一起画出来。图 9-2 中图像覆盖区域为训练区域并显示出了最终的分割结果。

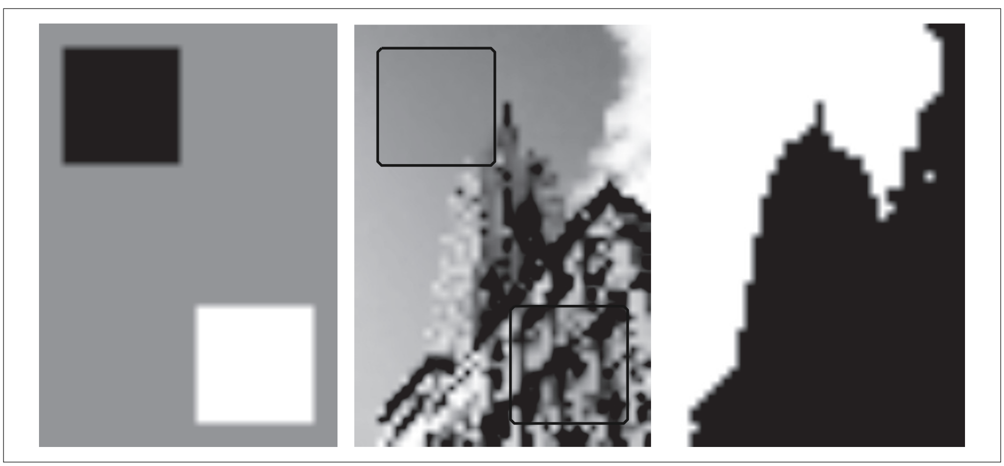


图 9-2：利用贝叶斯概率模型进行图割分割。图像降采样（downsample）到 54×38 大小。用于模型训练的标记图像（左）；在待分割图像上显示训练区域（中）；分割的结果（右）

变量 kappa（方程中的 K）决定了近邻像素间边的相对权重。改变 K 值分割的效果如图 9-3 所示；随着 K 值增大，分割边界将变得更平滑，并且细节部分也逐步丢失。你可以根据自己应用的需要及想要获得的结果类型来选择合适的 K 值。

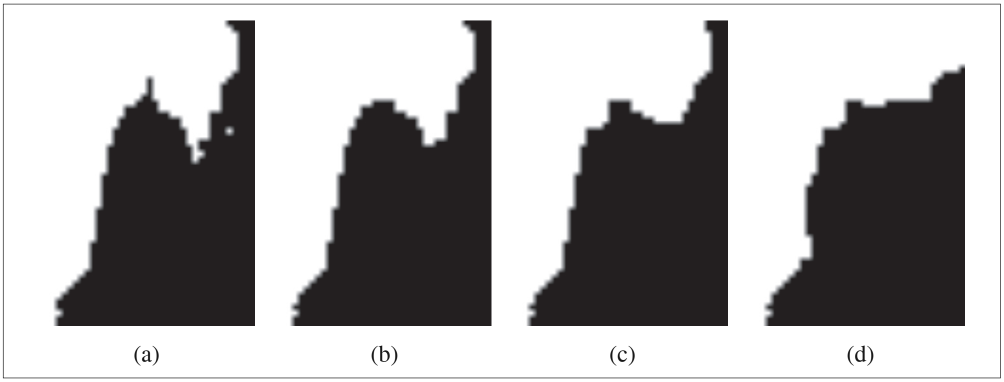


图 9-3：改变像素相似性和类概率之间相对权重对分割结果的影响，采用的分割方法与图 9-2 相同：（a）*k*=1，（b）*k*=2，（c）*k*=5，（d）*k*=10

9.1.2 用户交互式分割

利用一些方法可以将图割分割与用户交互结合起来。例如，用户可以在一幅图像上为前景和背景提供一些标记。另一种方法是利用边界框（bounding box）或“lasso”工具选择一个包含前景的区域。

让我们来看最后一个例子，其中用到了来自微软剑桥研究院 Grab Cut 数据集的一些图像，详见文献 [27] 和附录 B.5。

这些图像还提供了用来评价分割性能的真实标记，还有模拟用户选择矩形图像区域或用“lasso”之类的工具来标记前景和背景的标注信息。我们可以利用这些用户提供的输入来得到训练数据，并以用户输入为导向用切割对图像进行分割。

将用户输入编码成具有下面意义的位图图像：

像素值	意义
0, 64	背景
128	未知
255	前景

这里给出一个完整的示例代码，它会载入一幅图像及对应的标注信息，然后将其传递到我们的图割分割路径中：

```
from scipy.misc import imresize
import graphcut

def create_msr_labels(m,lasso=False):
```

```
""" 从用户的注释中创建用于训练的标记矩阵 """

labels = zeros(im.shape[:2])

# 背景
labels[m==0] = -1
labels[m==64] = -1

# 前景
if lasso:
  labels[m==255] = 1
else:
  labels[m==128] = 1

return labels


# 载入图像和注释图
im = array(Image.open('376043.jpg'))
m = array(Image.open('376043.bmp'))

# 调整大小
scale = 0.1
im = imresize(im,scale,interp='bilinear')
m = imresize(m,scale,interp='nearest')

# 创建训练标记
labels = create_msr_labels(m,False)

# 用注释创建图
g = graphcut.build_bayes_graph(im,labels,kappa=2)

# 图割
res = graphcut.cut_graph(g,im.shape[:2])

# 去除背景部分
res[m==0] = 1
res[m==64] = 1

# 绘制分割结果
figure()
imshow(res)
gray()
xticks([])
yticks([])
savefig('labelplot.pdf')
```

首先，我们定义一个辅助函数用以读取这些标注图像，格式化这些标注图像便于将其传递给背景和前景训练模型函数，矩形框中只包含背景标记。在本例中，我们设置前景训练区域为整个“未知的”区域（矩形内部）。下一步我们创建图并分割。由于有用户输入，所以我们移除那些在标记背景区域里有任何前景的结果。最后，我们绘制出分割结果，并通过设置这些勾选标记到一个空列表来移去这些勾选标记。这样我们就可以得到一个很好的边框（否则，图像中的边界在黑白图中很难看到）。

图 9-4 显示了利用 RGB 向量作为原始图像的特征进行分割的一些结果，一个下采样掩膜和下采样分割结果。右边的图像是通过上面的脚本生成的图线。

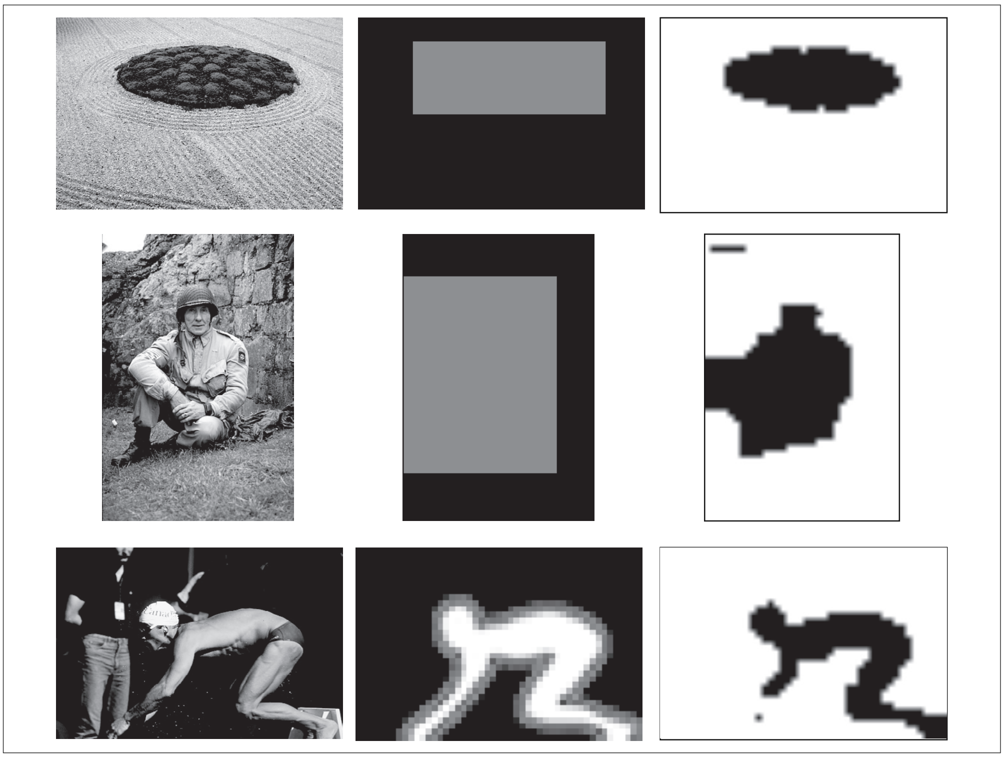


图 9-4：利用 Grab cut 数据集中的图像进行图割分割的结果：经过下采样后的原始图像（左）；用于训练的掩膜（中间）；（右）将用 RGB 像素值作为特征向量进行分割后的结果

9.2 利用聚类进行分割

上一节的图割问题通过在图像的图上利用最大流 / 最小割找到了一种离散解决方案。在本节，我们将看到另外一种分割图像图的方法，即基于谱图理论的归一化分割算法，它将像素相似和空间近似结合起来对图像进行分割。

该方法来自定义一个分割损失函数，该损失函数不仅考虑了组的大小而且还用划分的大小对该损失函数进行“归一化”。该归一化后的分割公式将方程（9.1）中分割损失函数修改为：

$$E_{ncut} = \frac{E_{cut}}{\sum_{i \in A} w_{ix}} + \frac{E_{cut}}{\sum_{j \in B} w_{jx}}$$

A 和 B 表示两个割集，并在图中分别对 A 和 B 中所有其他节点（函数进行“归一化”这里指图像像素）的权重求和相加，相加求和项称为 association。对于那些像素与其他像素具有相同连接数的图像，它是对划分大小的一种粗糙度量方式。文献 [32] 介绍了上面的损失函数与寻找极小值算法。该算法是针对两类分割问题衍生出来的，将在下面进行讲解。

定义 $\boldsymbol{W}$ 为边的权重矩阵，矩阵中的元素 w_{ij} 为连接像素 i 和像素 j 边的权重。$\boldsymbol{D}$ 为对 $\boldsymbol{W}$ 每行元素求和后构成的对角矩阵，即 $\boldsymbol{D}=diag(d_i)$，$d_i = \sum_j w_{ij}$（和 6.3 节中一样）。归一化分割可以通过最小化下面的优化问题而求得：

$$\min_{\mathbf{y}} \frac{\mathbf{y}^T(\boldsymbol{D}-\boldsymbol{W})\mathbf{y}}{\mathbf{y}^T\boldsymbol{D}\mathbf{y}}$$

向量 $\boldsymbol{y}$ 包含的是离散标记，这些离散标记满足对于 b 为常数 $y_i \in \{1, -b\}$（即 $\boldsymbol{y}$ 只可以取这两个值）的约束，$\boldsymbol{y}^T\boldsymbol{D}$ 求和为 0。由于这些约束条件，该问题不太容易求解 [1]。

然而，通过松弛约束条件并让 $\boldsymbol{y}$ 取任意实数，该问题可以变为一个容易求解的特征分解问题。这样求解的缺点是你需要对输出设定阈值或进行聚类，使它重新成为一个离散分割。

松弛该问题后，该问题便成为求解拉普拉斯矩阵特征向量问题：

$$\boldsymbol{L} = \boldsymbol{D}^{-1/2}\boldsymbol{W}\boldsymbol{D}^{-1/2}$$

正如谱聚类情形一样，现在的难点是如何定义像素间边的权重 w_{ij}。归一化割与谱聚类有很多相似之处，并且基础理论有所重叠，具体解释及细节参阅文献 [32]。

我们利用原始归一化割论文 [32] 中的边的权重，通过下式给出连接像素 i 和像素 j 的边的权重：

$$w_{ij} = \mathrm{e}^{-|I_i - I_j|^2/\sigma_g}\,\mathrm{e}^{-|x_i - x_j|^2/\sigma_d}.$$

式中第一部分度量像素 I_i 和 I_j 之间的像素相似性，$I_i(I_j)$ 定义为 RGB 向量或灰度值；

注 1：事实上，这是一个 NP 问题。

第二部分度量图像中 $\mathbf{x}_i$ 和 $\mathbf{x}_j$ 的接近程度，$x_i(x_j)$ 定义为每个像素的坐标矢量，缩放因子 σ_g 和 σ_d 决定了相对尺度和每一部件趋近 0 的快慢。

让我们看看这在代码中如何体现，将下面的函数添加到名为 ncut.py 的文件中：

```
def ncut_graph_matrix(im,sigma_d=1e2,sigma_g=1e-2):
  """ 创建用于归一化割的矩阵，其中 sigma_d 和 sigma_g 是像素距离和像素相似性的权重参数 """

  m,n = im.shape[:2]
  N = m*n

  # 归一化，并创建 RGB 或灰度特征向量
  if len(im.shape)==3:
    for i in range(3):
      im[:,:,i] = im[:,:,i] / im[:,:,i].max()
    vim = im.reshape((-1,3))
  else:
    im = im / im.max()
    vim = im.flatten()

  # x,y 坐标用于距离计算
  xx,yy = meshgrid(range(n),range(m))
  x,y = xx.flatten(),yy.flatten()

  # 创建边线权重矩阵
  W = zeros((N,N),'f')
  for i in range(N):
    for j in range(i,N):
      d = (x[i]-x[j])**2 + (y[i]-y[j])**2
      W[i,j] = W[j,i] = exp(-1.0*sum((vim[i]-vim[j])**2)/sigma_g) * exp(-d/sigma_d)

  return W
```

该函数获取图像数组，并利用输入的彩色图像 RGB 值或灰度图像的灰度值创建一个特征向量。由于边的权重包含了距离部件，对于每个像素的特征向量，我们利用 meshgrid() 函数来获取 x 和 y 值，然后该函数会在 N 个像素上循环，并在 $N \times N$ 归一化割矩阵 $\boldsymbol{W}$ 中填充值。

我们可以顺序分割每个特征向量或获取一些特征向量对它们进行聚类来计算分割结果。这里选择第二种方法，它不需要修改任意分割数也能正常工作。将拉普拉斯矩阵进行特征分解后的前 $ndim$ 个特征向量合并在一起构成矩阵 $\boldsymbol{W}$，并对这些像素进行聚类。下面函数实现了该聚类过程，可以看到，它和 6.3 节的谱聚类例子几乎是一样的：

```
from scipy.cluster.vq import *

def cluster(S,k,ndim):
  """ 从相似性矩阵进行谱聚类 """

  # 检查对称性
  if sum(abs(S-S.T)) > 1e-10:
    print 'not symmetric'

  # 创建拉普拉斯矩阵
  rowsum = sum(abs(S),axis=0)
  D = diag(1 / sqrt(rowsum + 1e-6))
  L = dot(D,dot(S,D))

  # 计算 L 的特征向量
  U,sigma,V = linalg.svd(L)

  # 从前 ndim 个特征向量创建特征向量
  # 堆叠特征向量作为矩阵的列
  features = array(V[:ndim]).T

  # K-means 聚类
  features = whiten(features)
  centroids,distortion = kmeans(features,k)
  code,distance = vq(features,centroids)

  return code,V
```

这里我们采用基于特征向量图像值的 K-means 聚类算法（细节参见 6.1 节）对像素进行分组。如果你想对该结果进行实验，可以采用任一聚类算法或分组准则。

现在我们可以利用该算法在一些样本图像上进行测试。下面的脚本展示了一个完整的例子：

```
import ncut
from scipy.misc import imresize

im = array(Image.open('C-uniform03.ppm'))
m,n = im.shape[:2]

# 调整图像的尺寸大小为 (wid,wid)
wid = 50
rim = imresize(im,(wid,wid),interp='bilinear')
rim = array(rim,'f')
```

```
# 创建归一化割矩阵
A = ncut.ncut_graph_matrix(rim,sigma_d=1,sigma_g=1e-2)

# 聚类
code,V = ncut.cluster(A,k=3,ndim=3)

# 变换到原来的图像大小
codeim = imresize(code.reshape(wid,wid),(m,n),interp='nearest')

# 绘制分割结果
figure()
imshow(codeim)
gray()
show()
```

因为 Numpy 的 `linanlg.svd()` 函数在处理大型矩阵时的计算速度并不够快（有时对于太大的矩阵甚至会给出不准确的结果），所以这里我们重新设定图像为一固定尺寸（在该例中为 50×50），以便更快地计算特征向量。在重新设定图像大小的时候我们采用了双线性插值法；因为不想插入类标记，所以在重新调整分割结果标记图像的尺寸时我们采用近邻插值法。注意，重新调整到原图像尺寸大小后第一次利用 reshape 方法将一维矩阵变换为（*wid*，*wid*）二维数组。

在该例中，我们用到了静态手势（Static Hand Posture）数据库（详见 8.1 节）的某幅手势图像，并且聚类数 *k* 设置为 3。分割结果如图 9-5 所示，取前 4 个特征向量。

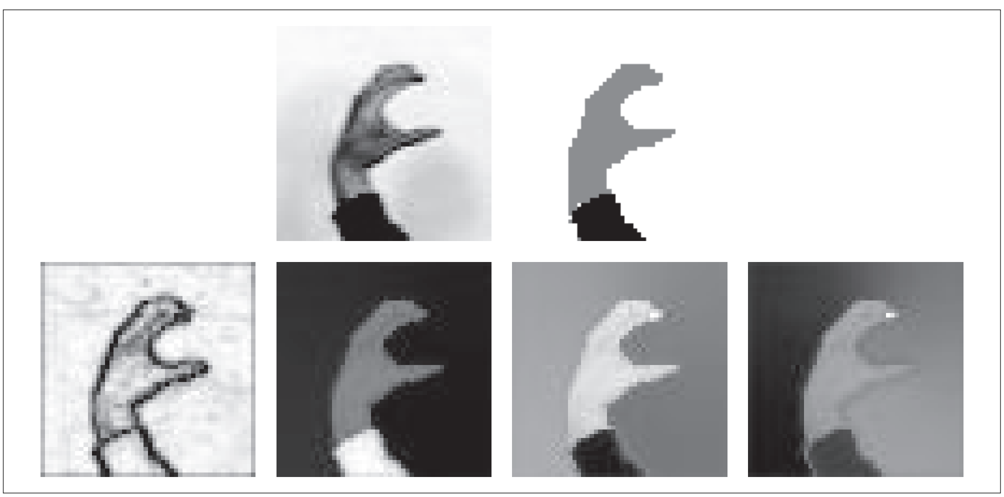


图 9-5：利用归一化分割算法分割图像：原始图像和三类分割结果（上）；图相似矩阵的前 4 个特征向量（下）

例子中的特征向量以数组 $\boldsymbol{V}$ 返回，可以通过下面的代码可视化为图像：

```
imshow(imresize(V[i].reshape(wid,wid),(m,n),interp='bilinear'))
```

它以原图像尺寸将特征向量 $\boldsymbol{i}$ 显示为一幅图像。

图 9-6 是一些利用上面脚本进行分割的示例。飞机图像来源于 Caltech 101 数据库中的飞机类。在这些例子中，我们保持参数 σ_g 和 σ_d 与上面一致，改变这两个参数的值可以得到更平滑、更规则的结果和不同的特征向量图像。我们将这个实验留给你完成。

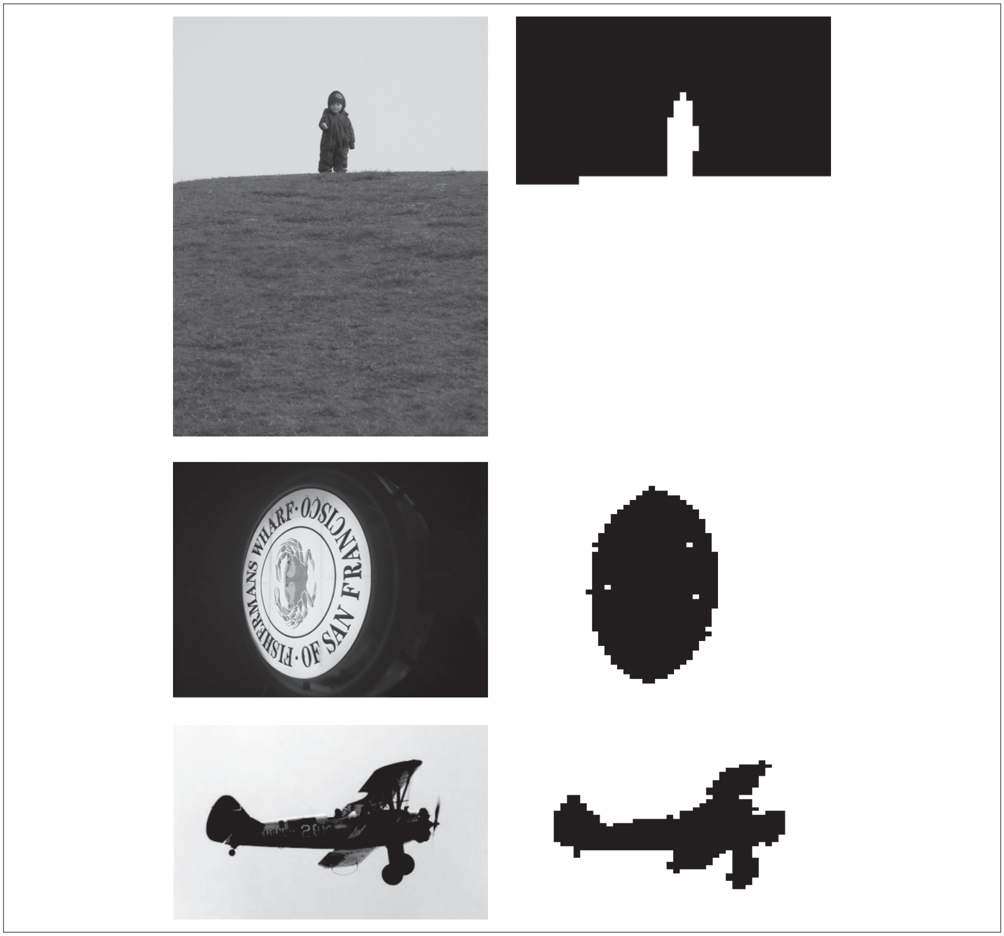


图 9-6：利用规范化分割算法对两类图像分割示例：原始图像（左边），分割结果（右边）

注意，即使对于这些相对简单的例子，对图像进行阈值处理不会给出相同的结果，对 RGB 值或灰度值进行聚类也一样；原因是它们都没有考虑像素的近邻。

9.3 变分法

在本书中有很多利用最小化代价函数或能量函数来求解计算机视觉问题的例子，如前面章节中在图中最小化割；我们同样可以看到诸如 ROF 降噪、K-means 和 SVM 的例子，这些都是优化问题。

当优化的对象是函数时，该问题称为变分问题，解决这类问题的算法称为变分法。我们看一个简单而有效的变分模型。

Chan-Vese 分割模型 [6] 对于待分割图像区域假定一个分片常数图像模型。这里我们集中关注两个区域的情形，比如前景和背景，不过这个模型也可以拓展到多区域，比如文献 [38] 中的例子。我们接下来对该模型进行描述。

如果我们用一组曲线 Γ 将图像分离成两个区域 Ω_1 和 Ω_2，如图 9-7 所示，分割是通过最小化 Chan-Vese 模型能量函数给出的：

$$E(\Gamma) = \lambda \,\mathrm{length}(\Gamma) + \int_{\Omega_1}(I - c_1)^2\mathrm{d}\mathbf{x} + \int_{\Omega_2}(I - c_2)^2\mathrm{d}\mathbf{x}$$

该能量函数用来度量与内部平均灰度常数 c_1 和外部平均灰度常数 c_2 的偏差。这里这两个积分是对各自区域的积分，分离曲线 Γ 的长度用以选择更平滑的方案。

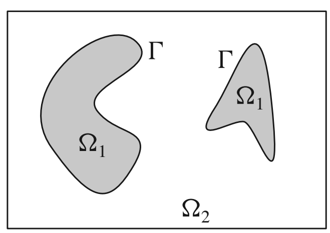


图 9-7：分片常数 Chan-Vese 分割模型

由分片常数图像 $U=\chi_1 c_1+\chi_2 c_2$，我们可以将上式重写为：

$$E(\Gamma) = \lambda \frac{|c_1 - c_2|}{2}\int |\nabla U|\mathrm{d}\mathbf{x} + \| I - U \|^2$$

χ_1 和 χ_2 是两区域 Ω_1、Ω_2 的特征（指示）函数 [1]。该变换意义重大，要求一些并不需要理解的复杂数学运算，并且已超出本书的讲述范围。

注 1：区域内特征函数为 1，区域外特征函数为 0。

如果用 $\lambda|c_1-c_2|$ 替换 ROF 方程 (1.1) 中的 λ，则该方程与 ROF 方程 (1.1) 的形式一致；唯一的区别在于，我们 Chan-Vese 模型中在寻找一幅具有分片常数的图像 U。对 ROF 方案进行阈值处理可以给出一个好的极小值，感兴趣的读者可以在文献 [8] 中查阅细节。

最小化 Chan-Vese 模型现在转变成为设定阈值的 ROF 降噪问题：

```
import rof

im = array(Image.open('ceramic-houses_t0.png').convert("L"))
U,T = rof.denoise(im,im,tolerance=0.001)
t = 0.4 # 阈值

import scipy.misc
scipy.misc.imsave('result.pdf',U < t*U.max())
```

在该示例中，为确保得到足够的迭代次数，我们调低 ROF 迭代终止时的容忍阈值。图 9-8 显示了两幅难以分割图像的分割结果。

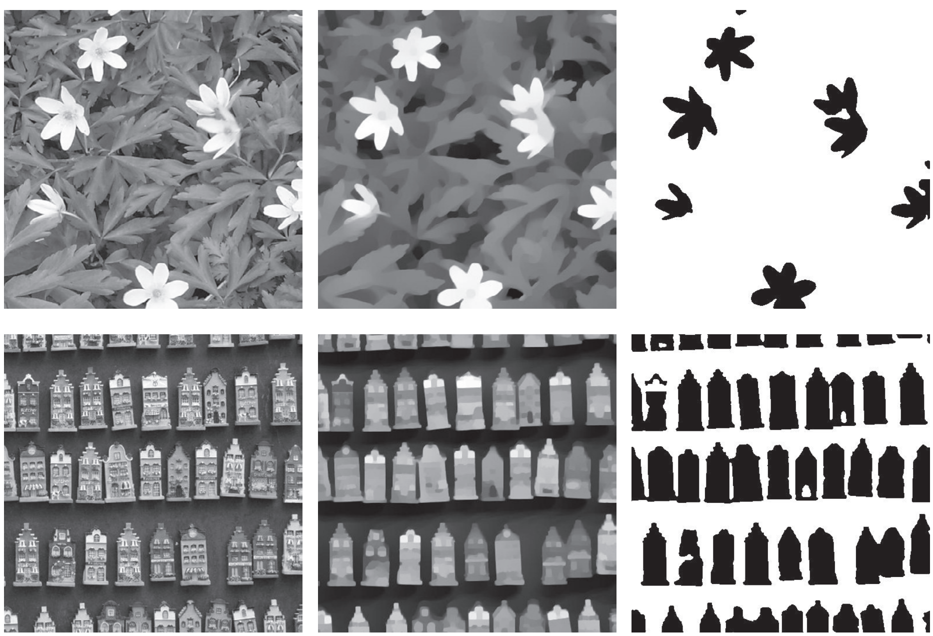


图 9-8：利用 ROF 降噪最小化 Chan-Vese 模型的一些图像分割示例：（a）为原始图像；（b）为经过 ROF 降噪后的图像；（c）为最终分割结果

练习

(1) 通过减少边的数目可以加速图割优化计算；文献 [16] 的 4.2 节描述了图的构建过程。对其进行实验并在图不同的大小下度量其差异，以及对分割时间的影响，并与我们之前用到的较简单图结构进行比较。

(2) 创建一个用户接口或模拟用户选择区域来进行图割，然后通过给一些较大的值设置权重尝试“硬编码”背景和前景。

(3) 在图割时将 RGB 特征向量换成别的特征描述子，你能以此改进分割结果吗？

(4) 用图割方法将当前分割结果用于训练下一个新的前景和背景模型，实现一种迭代分割方法。这样能够提高分割质量吗？

(5) 微软研究院 Grab Cut 数据集包含真实的分割图。实现一个可以度量分割误差的函数，并对不同的设置以及前面练习中的一些想法进行评估。

(6) 改变归一化的割边的权重参数，观察其如何影响特征向量图像，以及分割的结果。

(7) 在第一次归一化割的特征向量上计算图像梯度，合并这些梯度图像来检测图像中物体的轮廓。

(8) 在 Chan-Vese 分割算法中，对去噪后的图像在阈值上进行线性搜索。对于每个阈值，存储能量 $E(\Gamma)$，并选择具有最小值的分割结果。

第10章

OpenCV

本章概述如何通过 Python 接口使用流行的计算机视觉库 OpenCV。OpenCV 是一个 C++ 库，用于（实时）处理计算视觉问题。实时处理计算机视觉的 C++ 库，最初由英特尔公司开发，现由 Willow Garage 维护。OpenCV 是在 BSD 许可下发布的开源库，这意味着它对于学术研究和商业应用是免费的。OpenCV 2.0 版本对于 Python 的支持已经得到了极大的改善。下面，我们会讲解一些基本的例子并深入了解视频与跟踪。

10.1 OpenCV的Python接口

OpenCV 是一个 C++ 库，它包含了计算机视觉领域的很多模块。除了 C++ 和 C，Python 作为一种简洁的脚本语言，在 C++ 代码基础上的 Python 接口得到了越来越广泛的支持。目前，OpenCV 的 Python 接口仍在发展，不过并不是所有的 OpenCV 组件都提供了相应的 Python 接口，此处还有很多函数没有文档说明。因为该接口背后有一个很活跃的开发社区，所以这种现状很有可能得到改变。Python 接口文档说明见 http://opencv.willowgarage.com/documentation/python/index.html，安装说明参阅附录 A。

OpenCV 2.3.1 版本实际上提供了两个 Python 接口。旧的 `cv` 模块使用 OpenCV 内部数据类型，并且从 `NumPy` 使用起来可能需要一些技巧。新的 `cv2` 模块用到了 `NumPy` 数组，并且使用起来更加直观[1]，可以通过以下方式导入新的 `cv2` 模块：

```
import cv2
```

注 1：这两个模块的名称和位置可能会随时间改变。改变情况参阅在线文档。

而对于旧的 cv 模块可以通过以下方式导入：

```
import cv2.cv
```

本章中我们将关注 cv2 模块；注意这些名字的衍变、函数名称的改变以及在后来版本中的定义。现在，OpenCV 和 Python 接口在飞速发展中。

10.2 OpenCV基础知识

OpenCV 自带读取、写入图像函数以及矩阵操作和数学库。关于 OpenCV 的细节，有一本很好的书 [3]（只有 C++ 版）。我们现在来看一些基本的组件及其使用方法。

10.2.1 读取和写入图像

下面这个简短的例子会载入一张图像，打印出图像大小，对图像进行转换并保存为 .png 格式：

```
import cv2

# 读取图像
im = cv2.imread('empire.jpg')
h,w = im.shape[:2]
print h,w

# 保存图像
cv2.imwrite('result.png',im)
```

函数 imread() 返回图像为一个标准的 NumPy 数组，并且该函数能够处理很多不同格式的图像。如果你愿意，可以将该函数作为 PIL 模块读取图像的备选方案。函数 imwrite() 会根据文件后缀自动转换图像。

10.2.2 颜色空间

在 OpenCV 中，图像不是按传统的 RGB 颜色通道，而是按 BGR 顺序（即 RGB 的倒序）存储的。读取图像时默认的是 BGR，但是还有一些可用的转换函数。颜色空间的转换可以用函数 cvtColor() 来实现。例如，可以通过下面的方式将原图像转换成灰度图像：

```
im = cv2.imread('empire.jpg')
# 创建灰度图像
gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
```

在读取原图像之后，紧接其后的是 OpenCV 颜色转换代码，其中最有用的一些转换代码如下：

- cv2.COLOR_BGR2GRAY
- cv2.COLOR_BGR2RGB
- cv2.COLOR_GRAY2BGR

上面每个转换代码中，转换后的图像颜色通道数与对应的转换代码相匹配，比如对于灰度图像只有一个通道，对于 RGB 和 BGR 图像则有三个通道。最后的 cv2.COLOR_GRAY2BGR 将灰度图像转换成 BGR 彩色图像；如果你想在图像上绘制或覆盖有色彩的对象，CV2.COLOR_GRAY2BGR 是非常有用的，我们会在后面的例子中用到它。

10.2.3　显示图像及结果

我们来看一些用 OpenCV 处理图像的例子，以及怎样利用 OpenCV 绘制功能和窗口管理功能来显示结果。

第一个例子是从文件中读取一幅图像，并创建一个整数图像表示：

```
import cv2

# 读取图像
im = cv2.imread('fisherman.jpg')
gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

# 计算积分图像
intim = cv2.integral(gray)

# 归一化并保存
intim = (255.0*intim) / intim.max()
cv2.imwrite('result.jpg',intim)
```

读取图像后，将其转化为灰度图像，函数 `integral()` 创建一幅图像，该图像的每个像素值是原图上方和左边强度值相加后的结果；这对于快速评估特征是一个非常有用的技巧。OpenCV 的 `CascadeClassifier` 用到了积分图像，立足于 Viola 和 Jones 在文献 [39] 中引入的框架。在保存图像前，通过除以图像中的像素最大值将其归一化到 0 至 255 之间。图 10-1 显示了示例图像结果。

第二个例子从一个种子像素进行泛洪填充：

```
import cv2

# 读取图像
```

```
filename = 'fisherman.jpg'
im = cv2.imread(filename)
h,w = im.shape[:2]

# 泛洪填充
diff = (6,6,6)
mask = zeros((h+2,w+2),uint8)
cv2.floodfill(im,mask,(10,10), (255,255,0),diff,diff)

# 在 OpenCV 窗口中显示结果
cv2.imshow('flood fill',im)
cv2.waitKey()

# 保存结果
cv2.imwrite('result.jpg',im)
```

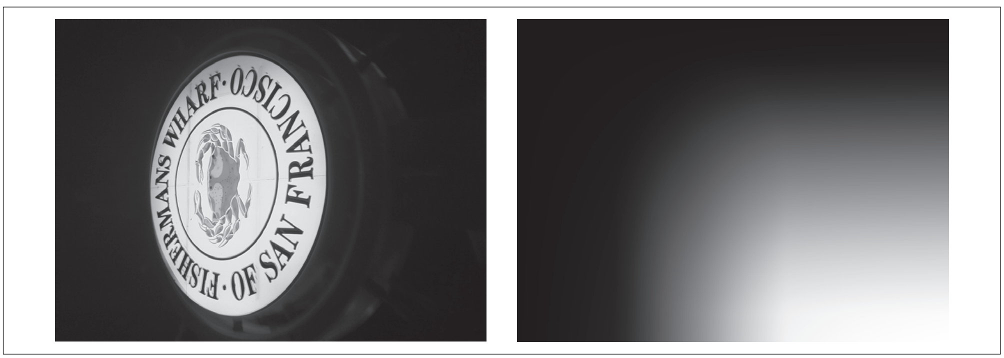


图 10-1：用 OpenCV 的 `integral()` 函数计算积分图像

在该例中，对图像应用泛洪填充并在 OpenCV 窗口中显示结果。`waitKey()` 函数一直处于暂停状态，直到有按键按下，此时窗口才会自动关闭。这里的 `floodfill()` 函数获取（灰度或彩色）图像、一个掩膜（非零像素区域表明该区域不会被填充）、一个种子像素以及新的颜色值来代替下限和上限阈值差的泛洪像素。泛洪填充以种子像素为起始，只要能在阈值的差异范围内添加新的像素，泛洪填充就会持续扩展。不同的阈值差异由元组 (R,G,B) 给出，结果如图 10-2 所示。

作为最后一个例子，我们来看 SURF 特征的提取，SURF 特征是 SIFT 特征的一个更快特征提取版，文献 [1] 引入。这里，我们也会展示怎样使用一些基本的 OpenCV 绘制命令：

```
import cv2

# 读取图像
im = cv2.imread('empire.jpg')
```

```
# 下采样
im_lowres = cv2.pyrDown(im)

# 变换成灰度图像
gray = cv2.cvtColor(im_lowres,cv2.COLOR_RGB2GRAY)

# 检测特征点
s = cv2.SURF()
mask = uint8(ones(gray.shape))
keypoints = s.detect(gray,mask)

# 显示结果及特征点
vis = cv2.cvtColor(gray,cv2.COLOR_GRAY2BGR)

for k in keypoints[::10]:
  cv2.circle(vis,(int(k.pt[0]),int(k.pt[1])),2,(0,255,0),-1)
  cv2.circle(vis,(int(k.pt[0]),int(k.pt[1])),int(k.size),(0,255,0),2)

cv2.imshow('local descriptors',vis)
cv2.waitKey()
```



图 10-2：彩色图像泛洪填充。在左上角用单个种子进行泛洪填充，右图高亮区域标出了所有填充了的像素

读取图像后，如果没有给定新尺寸，则用 pyrDown() 进行下采样，创建一个尺寸为原图像大小一半的新图像，然后将该图像转换为灰度图像，并传递到一个 SURF 关键点检测对象；掩膜决定了在哪些区域应用关键点检测器。在画出检测结果时，我们将灰度图像转换成彩色图像，并用绿色通道画出检测到的关键点。我们在每到第十个关键点时循环一次，并在中心画一个圆环，每一个圆环显示出关键点的尺度（大小）。绘图函数 circle() 获取一幅图像、图像坐标（仅整数坐标）元组、半径、绘图的颜色元组以及线条粗细（−1 是实线圆环）。图 10-3 显示了提取出来的 SURF 特征。

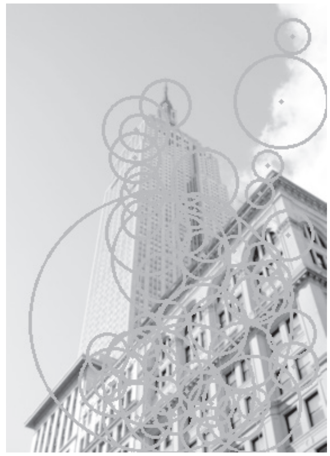


图 10-3：用 OpenCV 提取 SURF 特征并画出提取出来的 SURF 特征

10.3 处理视频

单纯使用 Python 来处理视频有些困难，因为需要考虑速度、编解码器、摄像机、操作系统和文件格式。目前还没有针对 Python 的视频库，使用 OpenCV 的 Python 接口是唯一不错的选择。在本节中，我们来看一些处理视频的基本示例。

10.3.1 视频输入

OpenCV 能够很好地支持从摄像头读取视频。下面给出了一个捕获视频帧并在 OpenCV 窗口中显示这些视频帧的完整例子：

```
import cv2

# 设置视频捕获
cap = cv2.VideoCapture(0)

while True:
  ret,im = cap.read()
  cv2.imshow('video test',im)
  key = cv2.waitKey(10)
  if key == 27:
    break
  if key == ord(' '):
    cv2.imwrite('vid_result.jpg',im)
```

捕获对象 VideoCapture 从摄像头或文件捕获视频。我们通过一个整数进行初始化，

该整数为视频设备的 id；如果仅有一个摄像头与计算机相连接，那么该摄像头的 id 为 0。read() 方法解码并返回下一视频帧，第一个变量 ret 是一个判断视频帧是否成功读入的标志，第二个变量则是实际读入的图像数组。函数 waitKey() 等待用户按键：如果按下的是 Esc 键（ASCII 码是 27）键，则退出应用；如果按下的是空格键，就保存该视频帧。

拓展上面的例子，将摄像头捕获的数据作为输入，并在 OpenCV 窗口中实时显示经模糊的（彩色）图像，我们只需对上面的例子做简单的修改：

```
import cv2

# 设置视频捕获
cap = cv2.VideoCapture(0)

# 获取视频帧，应用高斯平滑，显示结果
while True:
  ret,im = cap.read()
  blur = cv2.GaussianBlur(im,(0,0),5)
  cv2.imshow('camera blur',blur)
  if cv2.waitKey(10) == 27:
    break
```

每一视频帧都会被传递给 GaussianBlur() 函数，该函数会用高斯滤波器对传入的该帧图像进行滤波。这里，我们传递的是彩色图像，所以 Gaussian Blur() 函数会录入对彩色图像的每一个通道单独进行模糊。该函数需要为高斯函数设定滤波器尺寸（保存在元组中）及标准差；在本例中标准差设为 5。如果该滤波器尺寸设为 0，则它由标准差自动决定，显示出的结果与图 10-4 相似。

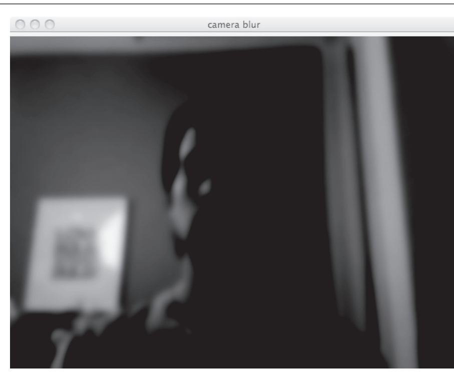


图 10-4：作者撰写本章时的一幅经过模糊的视频截图

以同样的方式从文件读取视频，不过我们调用 VideoCapture() 获取视频时是以文件名作为输入的：

```
capture = cv2.VideoCapture('filename')
```

10.3.2 将视频读取到NumPy数组中

使用 OpenCV 可以从一个文件读取视频帧，并将其转换成 NumPy 数组。下面是一个从摄像头捕获视频并将视频帧存储在一个 NumPy 数组中的例子：

```
import cv2

# 设置视频捕获
cap = cv2.VideoCapture(0)

frames = []
# 获取帧，存储到数组中
while True:
  ret,im = cap.read()
  cv2.imshow('video',im)
  frames.append(im)
  if cv2.waitKey(10) == 27:
    break
frames = array(frames)

# 检查尺寸
print im.shape
print frames.shape
```

上述代码将每一视频帧数组添加到列表末，直到捕获结束。最终得到的数组会有帧数、帧高、帧宽及颜色通道数（3 个），打印出的结果如下：

```
(480, 640, 3)
(40, 480, 640, 3)
```

这里共记录了 40 帧。类似上面将视频数据存储在数组中对于视频处理是非常有帮助的，比如计算帧间差异以及跟踪。

10.4 跟踪

跟踪是在图像序列或视频里对其中的目标进行跟踪的过程。

10.4.1 光流

光流是目标、场景或摄像机在连续两帧图像间运动时造成的目标的运动。它是图像在平移过程中的二维矢量场。作为一种经典并深入研究了的方法，它在诸如视频压缩、运动估计、目标跟踪和图像分割等计算机视觉中得到了广泛的应用。

光流法主要依赖于三个假设。

(1) 亮度恒定　图像中目标的像素强度在连续帧之间不会发生变化。
(2) 时间规律　相邻帧之间的时间足够短，以至于在考虑运行变化时可以忽略它们之间的差异。该假设用于导出下面的核心方程。
(3) 空间一致性　相邻像素具有相似的运动。

在很多情况下这些假设并不成立，但是对于相邻帧间的小运动以及短时间跳跃，它还是一个非常好的模型。假设一个目标像素在 t 时刻亮度为 $I(x,y,t)$，在 $t+\delta t$ 时刻运动 $[\delta x,\delta y]$ 后与 t 时刻具有相同的亮度，即 $I(x,y,t)=I(x+\delta x,\ y+\delta y,\ t+\delta t)$. 对该约束用泰勒公式进行一阶展开并关于 t 求偏导可以得到光流方程：

$$\nabla I^T \boldsymbol{v} = -I_t$$

$\boldsymbol{v}=[u,v]$ 是运动矢量，I_t 是时间偏导。对于图像中那些单个的点，该方程是线性方程组。由于 $\boldsymbol{v}$ 包含两个未知变量，所以该方程是不可解的。通过强制加入空间一致性约束，则有可能获得该方程的解。在下面的 Lucas-Kanade 算法中，我们将看到怎样利用该假设。

OpenCV 包含了一些光流实现：用了块匹配的 `CalcOpticalFlowBM()`；用了文献 [15]（这些只存在于旧的 cv 模块）的 `CalcOpticalFlowHS()`；文献 [19] 中的空间金字塔 Lucas-Kanade 算法 `calcOpticalFlowPyrLK()`；以及基于文献 [10] 的 `calcOpticalFlowFarneback()`。最后一种被视为获取密集流场的最好方法之一。让我们看一个利用 `calcOpticalFlowFarneback()` 在视频中寻找运动矢量的例子（Lucas-Kanade 光流法在下一节讲述）。

运行下面的脚本：

```
import cv2

def draw_flow(im,flow,step=16):
  """ 在间隔分开的像素采样点处绘制光流 """

  h,w = im.shape[:2]
  y,x = mgrid[step/2:h:step,step/2:w:step].reshape(2,-1)
```

```
    fx,fy = flow[y,x].T

    # 创建线的终点
    lines = vstack([x,y,x+fx,y+fy]).T.reshape(-1,2,2)
    lines = int32(lines)

    # 创建图像并绘制
    vis = cv2.cvtColor(im,cv2.COLOR_GRAY2BGR)
    for (x1,y1),(x2,y2) in lines:
      cv2.line(vis,(x1,y1),(x2,y2),(0,255,0),1)
      cv2.circle(vis,(x1,y1),1,(0,255,0), -1)
    return vis


# 设置视频捕获
cap = cv2.VideoCapture(0)

ret,im = cap.read()
prev_gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

while True:
  # 获取灰度图像
  ret,im = cap.read()
  gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

  # 计算流
  flow = cv2.calcOpticalFlowFarneback(prev_gray,gray,None,0.5,3,15,3,5,1.2,0)
  prev_gray = gray

  # 画出流矢量
  cv2.imshow('Optical flow',draw_flow(gray,flow))
  if cv2.waitKey(10) == 27:
    break
```

在这个例子中，利用摄像头捕获图像，并对每个连续图像对进行光流估计。由 calcOpticalFlowFarneback() 返回的运动光流矢量保存在双通道图像变量 flow 中。除了需要获取需要获取前一帧和当前帧，该函数还需要一系列参数。如果有兴趣可以查找相关的文献。辅助函数 draw_flow() 会在图像均匀间隔的点处绘制光流矢量，它利用 OpenCV 的绘图函数 line() 和 circle()，并用变量 step 控制流样本的间距。最终的结果如图 10-5 所示：圆环网格表示流样本的位置，带有线条的流矢量显示了每个样本点是怎样运动的。

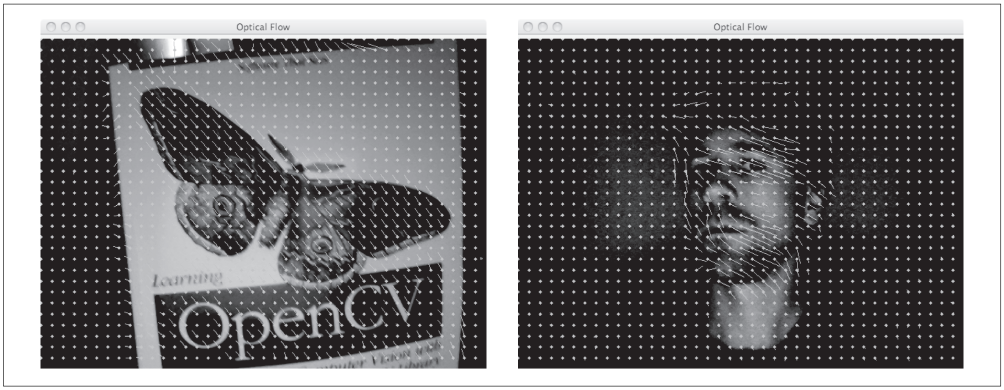


图 10-5：在书本平移和头部转动的视频上展示光流矢量（每隔 15 个像素采样一次）

10.4.2 Lucas-Kanade算法

跟踪最基本的形式是跟随感兴趣点，比如角点。对此，一次流行的算法是 Lucas-Kanade 跟踪算法，它利用了稀疏光流算法。

Lucas-Kanade 跟踪算法可以应用于任何一种特征，不过通常使用一些角点，比如 2.1 节的 Harris 角点。`goodFeaturesToTrack()` 函数采用 Shi 和 Tomasi 在文献 [33] 中设计的算法检测角点；角点是结构张量（Harris 矩阵）中有两个较大特征值的那些点，且更小的特征值要大于某个阈值。

如果基于每一个像素考虑，该光流方程组是欠定方程，即每个方程中含很多未知变量。利用相邻像素有相同运动这一假设，对于 n 个相邻像素，可以将这些方程写成如下系统方程：

$$\begin{bmatrix} \nabla I^T(\mathbf{x}_1) \\ \nabla I^T(\mathbf{x}_2) \\ \vdots \\ \nabla I^T(\mathbf{x}_n) \end{bmatrix} \boldsymbol{v} = \begin{bmatrix} I_x(\mathbf{x}_1) & I_y(\mathbf{x}_1) \\ I_x(\mathbf{x}_2) & I_y(\mathbf{x}_2) \\ \vdots & \vdots \\ I_x(\mathbf{x}_n) & I_y(\mathbf{x}_n) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = -\begin{bmatrix} I_t(\mathbf{x}_1) \\ I_t(\mathbf{x}_2) \\ \vdots \\ I_t(\mathbf{x}_n) \end{bmatrix}$$

该系统方程的优势是，现在方程的数目多于未知变量，并且可以用最小二乘法解出该系统方程。对于周围像素的贡献可以进行加权处理，使越远的像素影响越小；高斯权重是一种最常见的选择。将上面的矩阵变换成方程（2.2）的结构张量形式，可以得出以下关系：

$$\overline{\boldsymbol{M}_I} \boldsymbol{v} = -\begin{bmatrix} I_t(\mathbf{x}_1) \\ I_t(\mathbf{x}_2) \\ \vdots \\ I_t(\mathbf{x}_n) \end{bmatrix}，\text{或简记为 } \boldsymbol{Av=b}$$

该超定方程组可以用最小二乘法求解，得出运动矢量：

$$\boldsymbol{v} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\boldsymbol{b}$$

$\boldsymbol{v}$ 只有在 $\boldsymbol{A}^T\boldsymbol{A}$ 可逆时才是可解的，如果应用在 Harris 角点或 Shi-Tomasi 的“利于跟踪的好的特征 features to track”上，$\boldsymbol{A}^T\boldsymbol{A}$ 是可逆的。这就是 Lucas-Kanade 跟踪算法运行矢量怎样计算出来的全过程。

标准的 Lucas-Kanade 跟踪适用于小位移；为了能够处理较大位移，需要采用分层的方法。在该情形下，光流可以通过对图像由粗到精采样计算得到。这就是 OpenCV 函数 `calcOpticalFlowPyrLK()` 要做的事。

这些 Lucas-Kanade 函数包含在 OpenCV 中，让我们看看怎样利用这些函数建立一个 Python 跟踪类。创建名为 lktrack.py 的文件，向其添加下面的类和构造函数：

```
import cv2

# 一些常数及默认参数
lk_params = dict(winSize=(15,15),maxLevel=2,
        criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,10,0.03))

subpix_params = dict(zeroZone=(-1,-1),winSize=(10,10),
           criteria = (cv2.TERM_CRITERIA_COUNT | cv2.TERM_CRITERIA_EPS,20,0.03))

feature_params = dict(maxCorners=500,qualityLevel=0.01,minDistance=10)


class LKTracker(object):
  """  用金字塔光流 Lucas-Kanade 跟踪类 """

def __init__(self,imnames):
  """ 使用图像名称列表初始化 """

  self.imnames = imnames
  self.features = []
  self.tracks = []
  self.current_frame = 0
```

用一个文件名列表对跟踪对象进行初始化，变量 `features` 和 `tracks` 分别保存角点和对这些角点进行跟踪的位置，同时，我们也利用一个变量对当前帧进行跟踪。我们定义了三个字典变量用于特征提取、跟踪、和亚像素特征点的提炼。

在开始检测角点时，我们需要载入实际图像，并转换成灰度图像，提取“利用跟踪的好的特征”点。OpenCV 函数 `goodFeaturesToTrack()` 方法可以完成这一主要工

作。将下面的 detect_points() 添加到 LKTracker 类中：

```
def detect_points(self):
  """ 利用子像素精确度在当前帧中检测“利于跟踪的好的特征”（角点）"""

  # 载入图像并创建灰度图像
  self.image = cv2.imread(self.imnames[self.current_frame])
  self.gray = cv2.cvtColor(self.image,cv2.COLOR_BGR2GRAY)

  # 搜索好的特征点
  features = cv2.goodFeaturesToTrack(self.gray, **feature_params)

  # 提炼角点位置
  cv2.cornerSubPix(self.gray,features, **subpix_params)

  self.features = features
  self.tracks = [[p] for p in features.reshape((-1,2))]

  self.prev_gray = self.gray
```

上述代码用 cornerSubPix() 提炼角点位置，并保存在成员变量 features 和 tracks 中。需要注意的是，运行该函数会清除跟踪历史。

现在我们已经可以检测这些角点，接下来还需要对其进行跟踪。首先我们需要获得下一帧图像，然后应用 OpenCV 函数 calcOpticalFlowPyrLK() 找出这些点运动到哪里了，最后清除这些包含跟踪点的列表。下面的 track_points() 方法可以完成该过程：

```
def track_points(self):
  """ 跟踪检测到的特征 """

  if self.features != []:
    self.step() # 移到下一帧

    # 载入图像并创建灰度图像
    self.image = cv2.imread(self.imnames[self.current_frame])
    self.gray = cv2.cvtColor(self.image,cv2.COLOR_BGR2GRAY)

    #reshape() 操作，以适应输入格式
    tmp = float32(self.features).reshape(-1, 1, 2)

    # 计算光流
    features,status,track_error = cv2.calcOpticalFlowPyrLK(self.prev_gray,
                              self.gray,tmp,None,**lk_params)

    # 去除丢失的点
    self.features = [p for (st,p) in zip(status,features) if st]
```

```
        # 从丢失的点清楚跟踪轨迹
        features = array(features).reshape((-1,2))
        for i,f in enumerate(features):
          self.tracks[i].append(f)
        ndx = [i for (i,st) in enumerate(status) if not st]
        ndx.reverse()# 从后面移除
        for i in ndx:
          self.tracks.pop(i)

        self.prev_gray = self.gray
```

下面定义一个辅助函数 step()，用于移动到下一视频帧：

```
    def step(self,framenbr=None):
      """ 移到下一帧。如果没有给定参数，直接移到下一帧 """

      if framenbr is None:
        self.current_frame = (self.current_frame + 1) % len(self.imnames)
      else:
        self.current_frame = framenbr % len(self.imnames)
```

该方法会跳转到一个给定的视频帧，如果没有给定参数则直接跳转到下一帧。

最后，我们还希望能够用 OpenCV 窗口和绘图函数画出最终的跟踪结果。添加 draw() 方法到 LKTracker 类：

```
    def draw(self):
      """ 用 OpenCV 自带的画图函数画出当前图像及跟踪点，按任意键关闭窗口 """

      # 用绿色圆圈画出跟踪点
      for point in self.features:
        cv2.circle(self.image,(int(point[0][0]),int(point[0][1])),3,(0,255,0),-1)

      cv2.imshow('LKtrack',self.image)
      cv2.waitKey()
```

现在，我们用 OpenCV 函数实现了一个完整独立的跟踪系统。

1. 使用跟踪器

我们将该跟踪类用于真实的场景中。下面的脚本初始化一个跟踪对象，对视频序列进行角点检测、跟踪，并画出跟踪结果：

```
    import lktrack

    imnames = ['bt.003.pgm', 'bt.002.pgm', 'bt.001.pgm', 'bt.000.pgm']
```

```
# 创建跟踪对象
lkt = lktrack.LKTracker(imnames)

# 在第一帧进行检测，跟踪剩下的帧
lkt.detect_points()
lkt.draw()
for i in range(len(imnames)-1):
  lkt.track_points()
  lkt.draw()
```

每次画出一帧，并显示当前跟踪到的点，按任意键会转移到序列的下一帧。图 10-6 显示了牛津 corridor 序列（牛津对多视图数据集中的一个序列，参见 http://www.robots.ox.ac.uk/~vgg/data/data-mview.html）的前 4 幅图像的跟踪结果。



图 10-6：通过 LKTrack 类利用 Lucas-Kanade 算法进行跟踪

2. 使用发生器

将下面的方法添加到 LKTracker 类：

```
def track(self):
  """ 发生器，用于遍历整个序列 """

  for i in range(len(self.imnames)):
    if self.features == []:
      self.detect_points()
    else:
      self.track_points()
  #  创建一份 RGB 副本
  f = array(self.features).reshape(-1,2)
  im = cv2.cvtColor(self.image,cv2.COLOR_BGR2RGB)
  yield im,f
```

上面的方法创建一个发生器，可以使遍历整个序列并将获得的跟踪点和这些图像以 RGB 数组保存，以方便画出跟踪结果。将它用于经典的牛津“dinosaur”序列（也来源于上面提到的多视图数据集），并画出这些点及这些点的跟踪结果，代码如下：

```
import lktrack

imnames = ['viff.000.ppm', 'viff.001.ppm',
    'viff.002.ppm', 'viff.003.ppm', 'viff.004.ppm']

# 用 LKTracker 发生器进行跟踪
lkt = lktrack.LKTracker(imnames)
for im,ft in lkt.track():
  print 'tracking %d features' % len(ft)

# 画出轨迹
figure()
imshow(im)
for p in ft:
  plot(p[0],p[1],'bo')
for t in lkt.tracks:
  plot([p[0] for p in t],[p[1] for p in t])
axis('off')
show()
```

该发生器使前面定义的跟踪类的使用变得非常容易，并且完全向用户隐藏了 OpenCV 里的函数。该示例生成的结果如图 10-6 右下图和图 10-7 所示。

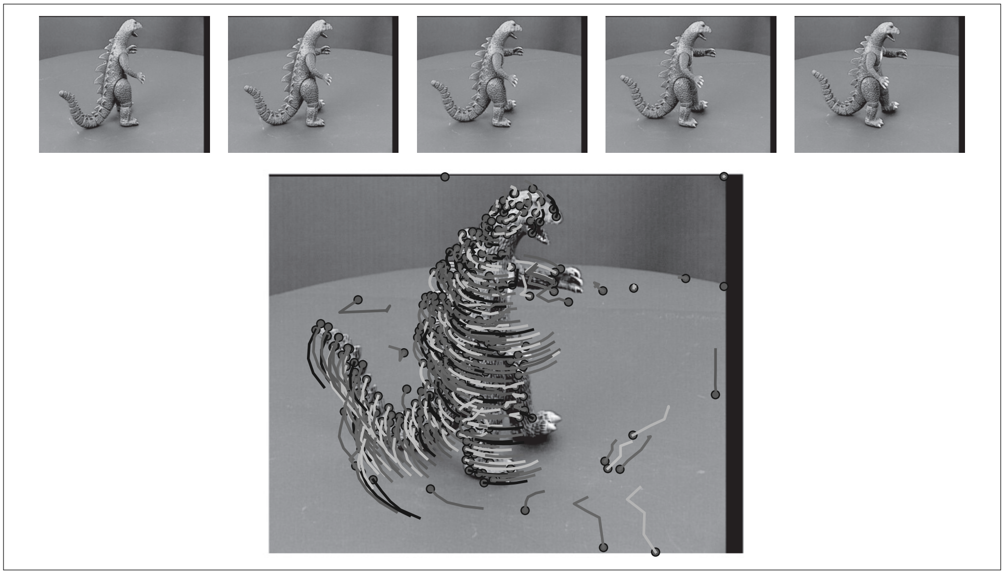


图 10-7：用 Lucas-Kanade 跟踪算法在转盘序列上跟踪并画出跟踪点的轨迹

10.5　更多示例

OpenCV 自带很多关于如何使用 Python 接口的有用示例。这些示例在子目录 samples/python2/ 中，用这些例子来熟悉 OpenCV 是一种非常好的方式。这里选择了一些来说明 OpenCV 的一些其他功能。

10.5.1　图像修复

对图像丢失或损坏的部分进行重建的过程叫做*修复*，既包括以复原为目的的对图像丢失数据或损坏部分进行恢复的算法，也包括在照片编辑应用程序中去除红眼或物体的算法。典型的例子是，图像的一个区域标记为“破损”，并需要利用余下的数据对该区域进行填补。

试着运行下面的命令：

```
$ python inpaint.py empire.jpg
```

运行上面的命令会打开一个交互窗口，在该窗口中你可以画一些需要修复的区域。最终修复的结果会在一个单独的窗口中显示出来，图 10-8 展示了一个示例。



图 10-8：用 OpenCV 进行图像修复的示例。左图显示了由用户标记的“破损”区域。右图显示了经过图像修复后的结果

10.5.2　利用分水岭变换进行分割

分水岭是一种可以用于分割（见图 10-9）的图像处理技术。图像可以看成是一幅有很多种子区域“淹没”后形成的拓扑地貌。由于梯度幅值图像在突出的边缘有脊，而且分割通常在这些边缘处停止，所以通常会用到梯度幅值图像。



图 10-9：用分水岭变换分割图像。左图是画有种子区域的输入图像，右图显示了分割结果，分割区域用不同的颜色覆盖

OpenCV 中的分水岭变换使用 Meyer [22] 的算法，可以使用下面的命令：

```
$ python watershed.py empire.jpg
```

该命令会打开一个交互窗口，你可以在该窗口中画一些种子区域作为输入。图 10-9 右图显示了用分水岭变换进行分割后的结果，在变换后的灰度图像中，不同颜色代表分割覆盖的区域。

10.5.3　利用霍夫变换检测直线

霍夫变换（http://en.wikipedia.org/wiki/Hough_transform）是一种用于在图像中寻找各种形状的方法，原理是在参数空间中使用投票机制。霍夫变换常用于在图像中寻找直线结构。在该情况下，可以在二维直线参数空间对相同的直线参数进行投票，将边缘和线段组合在一起。

OpenCV 可以利用该方法进行直线检测[1]，运行下面的命令：

```
$ python houghlines.py empire.jpg
```

它会给出图 10-10 中的两个窗口。第一个窗口显示了原图像进行灰度变换后的灰度图像，第二个显示了检测到的直线边缘，这些检测出来的直线在参数空间获得的投票最多。注意，这些直线通常是无限长的；如果你想在图像中找到线段的端点，可以使用边缘映射找到这些端点。



图 10-10：利用霍夫变换检测直线。左图是原图像经变换后的灰度图像，右图显示了检测到的直线

注 1：该示例当前在 /sample/python 文件夹中。

练习

(1) 用光流建立一个简单的手势识别系统。例如，你可以在绘图函数中对流采样，并将这些样本矢量作为输入。

(2) OpenCV 中有两个扭曲函数，`cv2.warpAffine()` 和 `cv2.warpPerspective()`。试着将它们用于第 3 章的一些例子中。

(3) 在图 10-7 的那些牛津“dinosaur”图像上用泛洪填充函数做背景减除，创建新的图像，将恐龙放在不同的颜色背景中或不同的图像中。

(4) OpenCV 有一个函数 `cv2.findChessboardCorners()`，它能够自动找到棋盘格的角点。使用此函数及 `cv2.calibrateCamera()` 函数来完成相机的校正。

(5) 如果你有两台摄像机，将它们安装在立体平台上，并以不同视频设备 id 用 `cv2.VideoCapture()` 捕获成对的立体图像。两台摄像机对应 `id` 分别为 0 和 1 开始，计算不同场景的景深图。

(6) 在 8.4 节数独 OCR 分类问题中使用 `cv2.HuMoments()` 提取的 Hu 不变矩作为特征，看看分类的效果如何。

(7) OpenCV 中有一个 Grab Cut 分割算法，在 9.1 节微软研究院 Grab Cut 数据集上用 `cv2.grabCut()` 函数进行图像分割。与我们在例子中用到的低分辨率分割相比，你应该会获得更好的分割结果。

(8) 修改 Lucas-Kanade 跟踪类，使其能够将一个视频文件作为输入，并写一个脚本，在帧与帧之间进行点跟踪，在每隔 *k* 帧时检测新点。

附录A
安装软件包

下面是本书中用到软件包的简要安装说明，基于撰写这本书时的最新版本。因为情况会因时间发生变化（网址变化！），所以如果以下说明过时，请检查各项目网站寻求帮助。

除了对各软件具体的说明，Python 的 easy_install 往往可以在大多数平台上使用。如果你遇到安装说明问题，easy_install 值得一试，详见 http://packages.python.org/distribute/easy_install.html。

A.1 NumPy和SciPy

安装 NumPy 和 SciPy 有一点不同，这取决于你的操作系统。请按照下面相应的说明安装，而大多数平台上的现行版本是 2.0（NumPy）和 0.11（SciPy）。目前可以在所有主要平台上运行的一个程序包是 Enthought 的 EPD Free，它是商业 Enthought 发行版的一个免费轻量版本，参见 http://enthought.com/products/epd_free.php。

A.1.1 Windows

安装 NumPy 和 SciPy 最简单的方法是在 http://www.scipy.org/Download 下载并安装二进制版本。

A.1.2 Mac OS X

最新版本的 Mac OS X（10.7.0 [Lion] 及以上）预装了 NumPy。

安装 NumPy 和 SciPy 到 Mac OS X 的一个简单方法是使用 SUPERPACK（https://github.com/fonnesbeck/ScipySuperpack）；这种方法也适用于 Matplotlib。

另一种方法是使用包管理系统 MacPorts（http://www.macports.org/）。除了下面的方法，这同样适用于 Matplotlib。

如果这些都不成功，该项目网页还列出了其他方法（http://scipy.org/）。

A.1.3 Linux

安装要求你有计算机的管理员权限。一些发行版预装了 NumPy，另一些则没有。NumPy 和 SciPy 都易于通过安装包内置的处理程序安装（例如 Ubuntu 的 Synaptic）。除了下面的方法，你也可以使用包处理程序安装 Matplotlib。

A.2 Matplotlib

这里是 Matplotlib 的安装说明，以防你在 NumPy/SciPy 的安装中没有安装 Matplotlib。Matplotlib 可以从 http://matplotlib.sourceforge.net/ 免费获取。点击 download（下载）链接，为你的系统和 Python 发行版下载最新版本的安装程序。目前最新的版本是 1.1.0。

你也可以下载源代码并解压，从命令行运行：

```
$ python setup.py install
```

应该一切正常。不同系统的一般安装提示可以参阅 http://matplotlib.sourceforge.net/users/installing.html，上述安装过程应该适用于大多数平台和 Python 版本。

A.3 PIL

PIL，即 Python 图像库，可在 http://www.pythonware.com/products/pil/ 获取。最新免费版本是 1.1.7。下载源代码包，解压。在解压后的文件夹中，从命令行运行：

```
$ python setup.py install
```

如果你想使用 PIL 保存图像，需要有 JPEG（libjpeg）和 PNG（zlib）支持。如果你遇到任何问题，请参阅 README 文件或 PIL 网站。

A.4 LibSVM

最新版本是 3.1（2011 年 4 月发布）。请从 LibSVM 网站（http://www.csie.ntu.edu.tw/~cjlin/libsvm/）下载 zip 文件，并解压（将创建目录 libsvm-3.1）。从终端进入该目录，输入 make：

```
$ cd libsvm-3.0
$ make
```

然后进入 python 目录，同样输入 make：

```
$ cd python/
$ make
```

上面 就是你所要做的。为了测试安装是否成功，在命令行启动 Python，尝试：

```
import svm
```

作者为使用 LibSVM [7] 撰写了实用指南。对于初学者来讲，这是一个很好的开始。

A.5 OpenCV

安装 OpenCV 有些不同，这取决于你的操作系统。按照下面相应的说明进行安装。

为检查安装是否成功，启动 Python 并尝试 http://opencv.willowgarage.com/documentation/python/cookbook.html 上的示例。对于如何使用 OpenCV 与 Python ，在线 OpenCV 的 Python 参考指南提供了更多的例子和细节，参见 http://opencv.willowgarage.com/documentation/python/index.html。

A.5.1 Windows 和 Unix

在 SourceForge 库里，有 Windows 和 Unix 的安装程序，参见 http://sourceforge.net/projects/opencvlibrary/。

A.5.2 Mac OS X

Mac OS X 的支持有所欠缺，但在不断改进。如 OpenCV 的 wiki 描述（http://opencv.willowgarage.com/wiki/InstallGuide），有几种方法可以从源代码进行安装，如果你使用 MacPorts 软件包管理器来安装 Python、Numpy 和 SciPy 或 Matplotlib，它会是一个不错的选择，可以这样从源代码安装 OpenCV：

```
$ svn co https://code.ros.org/svn/opencv/trunk/opencv
$ cd opencv/
```

```
$ sudo cmake -G "Unix Makefiles" .
$ sudo make -j8
$ sudo make install
```

如果你建立了所有的依赖关系，一切都应该正确建立并安装。如果你得到一个这样的错误：

```
import cv2
Traceback (most recent call last):
  file "", line 1, in
ImportError: No module named cv2
```

那么你需要将含 cv2.so 的目录添加到 PYTHONPATH。例如：

```
$ export PYTHONPATH=/usr/local/lib/python2.7/site-packages/
```

A.5.3 Linux

Linux 用户可以尝试发行版安装包（通常称为 opencv），或像 Mac OS X 一节中所描述的那样，从源代码安装。

A.6 VLFeat

安装 VLFeat，需要从 http://vlfeat.org/download.html（目前最新版本是 0.9.14）下载并解压缩最新的二进制软件包。把路径添加到你的环境或者把二进制文件复制到路径中的目录。二进制文件在 bin/ 目录，你可以结合自己的平台选择子目录。

VLFeat 命令行二进制文件的使用描述在 src/ 子目录。你也可以在 http://vlfeat.org/man/man.html 找到在线的说明文档。

A.7 PyGame

PyGame 可以从 http://www.pygame.org/download.shtml 下载，最新版本是 1.9.1。最简单的方法是获取与系统和 Python 版本相应的二进制安装包。

你也可以下载源代码，并在下载后的文件夹里从命令行中运行：

```
$ python setup.py install
```

A.8 PyOpenGL

安装 PyOpenGL 最简单的方法是按照 PyOpenGL 网页（http://pyopengl.sourceforge.

net/）的建议从 http://pypi.python.org/pypi/PyOpenGL 下载安装包。获取最新版本，目前是 3.0.1。

在下载文件夹，和之前一样运行

```
$ python setup.py install
```

如果你遇到问题或需要依赖性信息等，可以在 http://pyopengl.sourceforge.net/documentation/installation.html 找到更多说明文档。http://pypi.python.org/pypi/PyOpenGL-Demo 有一些很好的入门演示脚本。

A.9 Pydot

首先安装依赖关系，GraphViz 和 Pyparsing。转到 http://www.graphviz.org/，为你的平台下载最新的 GraphViz 二进制包。安装文件会自动安装 GraphViz。

然后，转到 Pyparsing 项目主页 http://pyparsing.wikispaces.com/。下载页面为 http://sourceforge.net/projects/pyparsing/。获取最新版本（目前是 1.5.5），并解压到一个目录下。在命令行输入：

```
$ python setup.py install
```

最后，转到项目页面 http://code.google.com/p/pydot/，点击 download（下载）。从下载页面下载最新版本（目前是 1.0.4）。解压并再次输入：

```
$ python setup.py install
```

现在你应该能够将 pydot 导入你的 Python 会话中。

A.10 Python-graph

Python-graph 是一个操作图表的 Python 模块，包含很多有用的算法，如遍历、最短路径、网页排名和最大流量；最新的版本是 1.8.1，可以在项目网站 http://code.google.com/p/pythongraph/ 上找到。如果你的系统上有 easy_install，最简单的方法是：

```
$ easy_install python-graph-core
```

或者，在 http://code.google.com/p/python-graph/downloads/list 下载源代码并运行：

```
$ python setup.py install
```

要编写并可视化图形（使用的 DOT 语言），你需要 python-graphdot，它可以下载或

使用 easy_install 安装：

```
$ easy_install python-graph-dot
```

Python-graph-dot 依赖于 pydot，如上所示。文档（HTML 格式）在 docs/ 文件夹中。

A.11 Simplejson

Simplejson 是 JSON 模块的独立维护版本，适合最 Python 新版（2.6 或更高版本）。两个模块的语法相同，但 simplejson 更优，并且能发挥更好的性能。

在项目页面 https://github.com/simplejson/simplejson 单击 Download 按钮。然后在 Download Packages（下载包）区域（目前是 2.1.3）选择最新版本。解压文件夹，在命令行中输入：

```
$ python setup.py install
```

一切 OK 了。

A.12 PySQLite

PySQLite 是一个为 Python 绑定的 SQLite。SQLite 是一个基于磁盘的轻量级数据库，可以使用 SQL 查询，并且易于安装和使用。最新的版本是 2.6.3，详见项目网站，http://code.google.com/p/pysqlite/。

从 http://code.google.com/p/pysqlite/downloads/list 下载文件并解压到一个文件夹，从命令行运行：

```
$ python setup.py install
```

A.13 CherryPy

CherryPy（http://www.cherrypy.org/）是一个快速、稳定、轻量级的 Web 服务器，基于 Python 建立，使用面向对象模型。CherryPy 易于安装，只需从 http://www.cherrypy.org/wiki/CherryPyInstall 下载最新版本，最新的稳定版本是 3.2.0。解压并运行：

```
$ python setup.py install
```

安装后，在 cherrypy/tutorial/ 文件夹查看 CherryPy 简单的示例教程。这些例子会告诉你如何传递 GET / POST 变量，继承页面特性，上传和下载文件等。

附录B
图像集

B.1 Flickr

广受欢迎的照片分享网站 Flickr（http://flickr.com/）是计算机视觉研究者和爱好者的金矿。这是一个很好的资源，包含数以亿计的图像，其中许多有用户做了标记，可以用来获得训练数据或用真实的数据进行实验。Flickr 有一个 API 接口服务，使其可以上传、下载和注释图像（以及更多功能）。对 API 完整的描述参见 http://flickr.com/services/api/，其中还包含许多编程语言的配套组件，包括 Python。

让我们看看如何使用名为 flickrpy 的库，参见 http://code.google.com/p/flickrpy/。下载文件 flickr.py。你需要从 Flickr 获得一个 API 密钥使其正常工作，这些密钥对于非商业用途是免费，对于商业用途则另有要求。点击 Flickr API 页面的链接“Apply for a new API Key”（申请新 API 密钥），然后按指示进行。获得 API 密钥后，打开 flickr.py，用密钥替换下面的空字符串：

```
API_KEY = ' '
```

结果如下：

```
API_KEY = '123fbbb81441231123cgg5b123d92123'
```

让我们创建一个简单的命令行工具，用来下载具有特定标签的图片。添加以下代码到一个名为 tagdownload.py 的文件中：

```
iimport flickr
import urllib, urlparse
import os
import sys
if len(sys.argv)>1:
     tag = sys.argv[1]
else:
     print 'no tag specified'

# 下载图像数据
f = flickr.photos_search(tags=tag)
urllist = [] #  用于存储下载了什么的列表

# 下载图像
for k in f:
     url = k.getURL(size='Medium', urlType='source')
     urllist.append(url)
     image = urllib.URLopener()
     image.retrieve(url, os.path.basename(urlparse.urlparse(url).path))
     print 'downloading:', url
```

如果你想将 URL 列表写入一个文本文件，可以在末尾添加下面的代码：

```
# 将 url 的列表写入文件
fl = open('urllist.txt', 'w')
for url in urllist:
     fl.write(url+'\n')
fl.close()
```

在命令行输入：

```
$ python tagdownload.py goldengatebridge
```

你会得到 100 幅标记为“goldengatebridge”的最新图像。可以看到，我们选择了获取“中等”尺寸的图像。如果你想得到缩略图、原尺寸的图像或其他图像，这里有许多其他尺寸供选择；说明文档参见 Flickr 网站 http://flickr.com/api/。

这里我们只对下载图像感兴趣；需要身份验证的 API 调用过程略微复杂。查看 API 文档了解更多的关于设置身份验证会话的信息。

B.2 Panoramio

谷歌照片分享服务 Panoramio（http://www.panoramio.com/）是一个获取地理标记图像的好地方。该 Web 服务提供 API 以编程方式访问内容。API 的描述参见 http://

www.panoramio.com/api/。你可以得到网站小部件并使用 JavaScript 对象访问数据。要从该网站上下载图片，最简单的方法是调用 GET，例如：

```
http://www.panoramio.com/map/get_panoramas.php?order=popularity&set=public&
from=0&to=20&minx=-180&miny=-90&maxx=180&maxy=90&size=mediu
```

minx、miny、maxx 和 maxy 定义选择照片的地理区域（分别表示最小经度、最小纬度、最大经度和最大纬度）。响应将用 JSON 格式表示，显示如下：


```
{"count": 3152, "photos":
[{"upload_date": "02 February 2006", "owner_name": "***", "photo_id": 9439,
"longitude": -151.75, "height": 375, "width": 500, "photo_title": "***",
"latitude": -16.5, "owner_url": "http://www.panoramio.com/user/1600", "owner_id": 1600,
"photo_file_url": "http://mw2.google.com/mw-panoramio/photos/medium/9439.jpg",
"photo_url": "http://www.panoramio.com/photo/9439"},
{"upload_date": "18 January 2011", "owner_name": "***", "photo_id": 46752123,
"longitude": 120.52718600000003, "height": 370, "width": 500, "photo_title": "***",
"latitude": 23.327833999999999, "owner_url": "http://www.panoramio.com/user/2780232",
"owner_id": 2780232,
"photo_file_url": "http://mw2.google.com/mw-panoramio/photos/medium/46752123.jpg",
"photo_url": "http://www.panoramio.com/photo/46752123"},
{"upload_date": "20 January 2011", "owner_name": "***", "photo_id": 46817885,
"longitude": -178.13709299999999, "height": 330, "width": 500, "photo_title": "***",
"latitude": -14.310613, "owner_url": "http://www.panoramio.com/user/919358",
"owner_id": 919358,
"photo_file_url": "http://mw2.google.com/mw-panoramio/photos/medium/46817885.jpg",
"photo_url": "http://www.panoramio.com/photo/46817885"},
.
.
.
], "has_more": true}
```


使用 JSON 包，你可以获得 photo_file_url 字段的结果，见 2.3 节中的例子。

B.3　牛津大学视觉几何组

牛津大学视觉几何研究组在 http://www.robots.ox.ac.uk/~vgg/data/ 上公布有很多的数据集。在本书中，我们使用了一些多视图的数据集，例如“Merton1”、“Model House”、“dinosaur”和“corridor”序列，这些数据（某些包括摄像机矩阵和点跟踪）下载地址为 http://www.robots.ox.ac.uk/~vgg/data/data-mview.html。

B.4　肯塔基大学识别基准图像

肯塔基大学基准图像集，亦称“ukbench”集，是一个包含 2550 组图片的图像集。

该数据集中，每组图像共 4 幅，包含同一对象或同一场景下的不同视角。这是一个测试目标识别和图像检索算法一个很好的图像集。数据集可以在 http://www.vis.uky.edu/~stewe/ukbench/ 下载（全套约 1.5 GB），详见文献 [23]。

在本书中，我们使用一个较小的子集，只含有前 1000 幅图片。

B.5 其他

B.5.1 Prague Texture Segmentation Datagenerator与基准

该数据集在分割那一章里面用到过，可以生成许多不同类型的纹理分割图像，参见 http://mosaic.utia.cas.cz/index.php。

B.5.2 微软剑桥研究院Grab Cut数据集

最初用于 Grab Cu 的论文 [27]，该图像集提供带有用户注释的分割图像。该数据集和一些论文参见 http://research.microsoft.com/en-us/um/cambridge/projects/visionimagevideoediting/segmentation/grabcut.html。数据集中的原始图像现在是伯克利分割数据集（http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/）中的一部分。

B.5.3 Caltech 101

这是一个经典的数据集，其中包含 101 类照片，可以用来测试目标识别算法，参见 http://www.vision.caltech.edu/Image_Datasets/Caltech101/。

B.5.4 静态手势数据库

这个来自 Sebastien Marcel 的数据集与其他几个手势数据集可在 http://www.idiap.ch/resource/gestures/ 下载。

B.5.5 Middlebury Stereo数据集

这些数据集用于基准立体算法，可在 http://vision.middlebury.edu/stereo/data/ 下载。每个立体像对都带有真实深度图像来方便比较结果。

附录C

图片来源

在本书中，我们从 Web 服务充分享用了公开可用的数据和图像，详见附录 B。非常感谢这些数据集背后研究人员的贡献。

一些反复出现的图像例子属于作者自己。你可以在创作共用署名 3.0 许可证（CC 3.0，http://creativecommons.org/licenses/by/3.0/）下免费使用这些图片，如引用本书。

这些图片是：

- 用于本书几乎所有例子的帝国大厦图像；
- 在图 1-7 中的低对比度图像；
- 用于图 2-2、图 2-5、图 2-6 和图 2-7 的特征匹配例子；
- 用于图 9-6、图 10-1 和图 10-2 的渔人码头标志；
- 用于图 6-4 和图 9-6 的山顶小男孩；
- 在图 4-3 中用于校准的书的图像；
- 用于图 4-4、图 4-5 和图 4-6 中的 O’Reilly 开源书的两幅图片。

C.1　来自Flickr的图像

我们在创作共同署名 2.0 通用许可证（CC2.0）使用了一些来自 Flickr 的图像（http://creativecommons.org/licenses/by/2.0/deed.en）下，非常感谢那些摄影师的贡献。

来自 Flickr 的图像（例子中使用的名称，不是原始文件名）：

- billboard_for_rent.jpg，来自 @striatic，http://flickr.com/photos/striatic/21671910/，用于图 3-2；
- blank_billboard.jpg， 来 自 @mediaboytodd，http://flickr.com/photos/23883605@N06/2317982570/，用于图 3-3；
- beatles.jpg，来自 @oddsock，http://flickr.com/photos/oddsock/82535061/，用于图 3-2 和图 3-3；
- turningtorso1.jpg， 来 自 @rutgerblom，http://www.flickr.com/photos/rutgerblom/2873185336/，用于图 3-5；
- sunset_tree.jpg，来自 @jpck，http://www.flickr.com/photos/jpck/3344929385/，用于图 3-5。

C.2 其他图像

- 用于图 3-6、图 3-7 和图 3-8 的人脸图像是由 J. K. Keller 提供的。 眼睛和嘴巴的注释是作者加的。
- 用于图 3-9、图 3-11 和图 3-12 的隆德大学建筑图片来自隆德大学数学成像组的一个数据集。摄影师很可能是 Magnus Oskarsson。
- 用于图 4-6 的玩具飞机三维模型来自 Gilles Tran（署名创作共同许可证）。
- 用于图 5-7 和图 5-8 的恶魔图像由 Carl Olsson 提供。
- 用于图 1-8、图 6-2、图 6-3、图 6-7 和图 6-8 的字体数据集由马丁 Martin Solli 提供。
- 用于图 8-6、图 8-7 和图 8-8 的数独图像由 Martin Byröd 提供。

C.3 插图

本书图 5-1 中对极几何的解释来自 Klas Josephson 的图解，并在本书中稍作了修改。

参考文献

[1] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. SURF: Speeded up robust features. In *European Conference on Computer Vision*, 2006.

[2] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:2001, 2001.

[3] Gary Bradski and Adrian Kaehler. *Learning OpenCV*. O'Reilly Media Inc., 2008.

[4] Martin Byröd. An optical Sudoku solver. In *Swedish Symposium on Image Analysis, SSBA*. *http://www.maths.lth.se/matematiklth/personal/byrod/papers/sudokuocr.pdf*, 2007.

[5] Antonin Chambolle. Total variation minimization and a class of binary mrf models. In *Energy Minimization Methods in Computer Vision and Pattern Recognition*, Lecture Notes in Computer Science, pages 136–152. Springer Berlin / Heidelberg, 2005.

[6] T. Chan and L. Vese. Active contours without edges. *IEEE Trans. Image Processing*, 10(2):266–277, 2001.

[7] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at *http://www.csie.ntu.edu.tw/~cjlin/libsvm*.

[8] D. Cremers, T. Pock, K. Kolev, and A. Chambolle. Convex relaxation techniques for segmentation, stereo and multiview reconstruction. In *Advances in Markov Random Fields for Vision and Image Processing*. MIT Press, 2011.

[9] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[10] Gunnar Farnebäck. Two-frame motion estimation based on polynomial expansion. In *Proceedings of the 13th Scandinavian Conference on Image Analysis*, pages 363–370, 2003.

[11] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications-of-the-ACM*, 24(6):381–95, 1981.

[12] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings Alvey Conference*, pages 189–192, 1988.

[13] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[14] Richard Hartley. In defense of the eight-point algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:580–593, 1997.

[15] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artifical Intelligence*, 17:185–203, 1981.

[16] Vladimir Kolmogorov and Ramin Zabih. What energy functions can be minimized via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:65–81, 2004.

[17] David G. Lowe. Object recognition from local scale-invariant features. In *International Conference on Computer Vision*, pages 1150–1157, 1999.

[18] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[19] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision, pages 674–679, 1981.

[20] Mark Lutz. *Learning Python*. O'Reilly Media Inc., 2009.

[21] Will McGugan. *Beginning Game Development with Python and Pygame*. Apress, 2007.

[22] F. Meyer. Color image segmentation. In *Proceedings of the 4th Conference on Image Processing and its Applications*, pages 302–306, 1992.

[23] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, pages 2161–2168, 2006.

[24] Travis E. Oliphant. *Guide to NumPy*. *http://www.tramy.us/numpybook.pdf*, 2006.

[25] M. Pollefeys, L. Van Gool, M. Vergauwen, F. Verbiest, K. Cornelis, J. Tops, and R. Koch. Visual modeling with a hand-held camera. *International Journal of Computer Vision*, 59(3):207–232, 2004.

[26] Marc Pollefeys. Visual 3d modeling from images—tutorial notes. Technical report, University of North Carolina–Chapel Hill. *http://www.cs.unc.edu/~marc/tutorial.pdf*

[27] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. Grabcut: Interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics*, 23: 309–314, 2004.

[28] L. I. Rudin, S. J. Osher, and E. Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D*, 60:259–268, 1992.

[29] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 2001.

[30] Daniel Scharstein and Richard Szeliski. High-accuracy stereo depth maps using structured light. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2003.

[31] Toby Segaran. *Programming Collective Intelligence*. O'Reilly Media, 2007.

[32] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22:888–905, August 2000.

[33] Jianbo Shi and Carlo Tomasi. Good features to track. In *1994 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94)*, pages 593–600, 1994.

[34] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3d. In *SIGGRAPH Conference Proceedings*, pages 835–846. ACM Press, 2006.

[35] Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. Bundle adjustment - a modern synthesis. In *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice*, ICCV '99, pages 298–372. Springer-Verlag, 2000.

[36] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. *http://www.vlfeat.org/*, 2008.

[37] Deepak Verma and Marina Meila. A comparison of spectral clustering algorithms. Technical report, 2003.

[38] Luminita A. Vese and Tony F. Chan. A multiphase level set framework for image segmentation using the mumford and shah model. *International Journal of Computer Vision*, 50:271–293, December 2002.

[39] Paul Viola and Michael Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.

[40] Marco Zuliani. Ransac for dummies. Technical report, Vision Research Lab, UCSB, 2011.

索引

A

B

C

D

E

F

G

Z

Python计算机视觉编程

想要了解计算机视觉的基本理论与算法？这本实践指南绝对会让你爱不释手。本书以大量简单明了的Python示例为依托，全面细致地介绍了对象识别、3D重建、立体成像、增强现实等技术。

书中没有枯燥的理论，而是结合代码示例讨论了计算机视觉的概念与应用。除了提供完整的示例代码以阐述计算机视觉编程的技术与方法，本书还从读者角度出发，提供了很多练习，以便巩固所学知识。只要有一定编程与数学基础，无论学生、研究人员，还是计算机视觉编程爱好者，本书都不容错过。

- 机器人导航、医学图像分析及其他计算机视觉应用技巧
- 图像映射与变换，包括纹理扭曲和全景创建
- 同一场景下多图像的3D重建
- 使用聚类方法，基于相似性或内容组织图像
- 构建有效的图像检索技巧，基于视觉内容搜索图像
- 用算法实现图像内容的分类和对象识别
- 通过Python接口访问流行的OpenCV库

"本书介绍了各种图像分析工具，是了解计算机视觉编程的'必备'读物。"

——James A. Cox
美国《中西部书评》
（*Midwest Book Review*）总编辑

Jan Erik Solem

瑞典隆德大学副教授（数学成像小组），Polar Rose公司创始人兼CTO，计算机视觉研究者，Python爱好者，技术图书作家，经常出席各种计算机视觉、图像分析、机器智能等国际会议并发表演讲。他主要关注3D重建、变分问题与优化、图像分割与识别、形状分析，有多年Python计算机视觉教学、研究和行业应用经验，技术博客为http://www.janeriksolem.net。另著有*Computing with Python: An Introduction to Python for Science and Engineering*一书。

封面设计：Karen Montgomery　马冬燕

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机/程序设计/Python

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

O'REILLY®
oreilly.com.cn

ISBN 978-7-115-35232-3

ISBN 978-7-115-35232-3
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks